Thesis for the degree of Doctor of Philosophy

A functional approach to heterogeneous computing in embedded systems

Markus Aronsson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2020

A functional approach to heterogeneous computing in embedded systems
Markus Aronsson
ISBN 978-91-7905-328-4

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 4795
ISSN 0346-718X

Department of Computer Science and Engineering
Chalmers University of Technology
SE–412 96 Göteborg, Sweden
Telephone + 46 (0) 31 – 772 1000

Typeset by the author using LaTeX.

Printed by Chalmers Reproservice
Göteborg, Sweden 2020

to my family

Abstract

Developing programs for embedded systems presents quite a challenge; not only should programs be resource efficient, as they operate under memory and timing constraints, but they should also take full advantage of the hardware to achieve maximum performance. Since performance is such a significant factor in the design of embedded systems, modern systems typically incorporate more than one kind of processing element to benefit from specialized processing capabilities. For such heterogeneous systems the challenge in developing programs is even greater.

In this thesis we explore a functional approach to heterogeneous system development as a means to address many of the modularity problems that are typically found in the application of low-level imperative programming for embedded systems. In particular, we explore a staged hardware software co-design language which we name Co-Feldspar and embed in Haskell. The staged approach enables designers to build their applications from reusable components and skeletons while retaining control over much of the generated source code. Furthermore, by embedding the language in Haskell we can exploit its type classes to write not only hardware and software programs, but also generic programs with overloaded instructions and expressions. We demonstrate the usefulness of the functional approach for co-design on a cryptographic example and signal processing filters, and benchmark software and mixed hardware-software implementations.

Co-Feldspar currently adopts a monadic interface, which provides an imperative functional programming style that is suitable for explicit memory management and algorithms that rely on a certain evaluation order. For algorithms that are better defined as pure functions operating on immutable values, we provide a signal and array language which extends a monadic language, like Co-Feldspar. These extensions permit a functional style of programming by composing high-level combinators. Our compiler transforms such high-level code into efficient programs with mutating code. In particular, we show how to execute an FFT safely in-place, and how to describe a FIR and IIR filter efficiently as streams.

Co-Feldspar's monadic interface is however quite invasive; not only is the burden of explicit memory management quite heavy on the user, it is also quite easy to shoot oneself in the foot. It is for these reasons that we also explore a dynamic memory management discipline that is based on regions but predictable enough to be of use for embedded systems. Specifically, this thesis introduces a program analysis which annotates values with dynamically allocated memory regions. By limiting our efforts to functional languages that target embedded software, we manage to define a region inference algorithm that is considerably simpler than traditional approaches.

Keywords: Functional programming, signal processing, region inference, hardware software co-design.

Acknowledgments

To my dearest, my fiancée Emma Bogren: because I owe it all to you.

My constant cheerleaders, that is, my parents Dag and Lena: I am forever grateful for their moral and emotional support, you were always keen to know what I was doing and how I was proceeding. Although I'm fairly certain you never fully grasped what my work was all about, you never wavered in their encouragement and enthusiastic inquires. I am also grateful to my little sister Caroline, who has supported me along the way, and her wonderful dog Alfons who never failed to brighten my day.

A very special gratitude goes out to my advisor Mary Sheeran, for her continuous help and support in my studies, for her never ending patience, guidance and immense knowledge. I send a heartfelt thanks to my co-workers Nick Smallbone and Ann Lillieström. My former co-worker Emil Axelsson is worthy of a special mention, for without his precious support I would not have been able to conduct much of my research.

I am also grateful to Anders Persson, Josef Svenningsson and Koen Claessen for their unfailing support and assistance. Their hard work, ideas and insights in the Feldspar project have proved a great source of inspiration in my research. Finally I express my gratitude to all my other colleagues at Chalmers, who make this a fantastic place to work.

Thank you all for your encouragement!

Markus Aronsson
Göteborg, June 2020

List of Publications

This thesis is based on the following appended papers:

Paper 1. Aronsson, Markus and Axelsson, Emil and Sheeran, Mary. *Stream Processing for Embedded Domain Specific Languages.* In: *Proceedings of the 26nd International Symposium on Implementation and Application of Functional Languages. 2014.*

This paper presents my extension of functional languages with support for signal processing. I am the lead author but Emil Axelsson wrote the majority of sections 3.1 and 3.2. The paper was awarded the Peter Landin award for best paper.

Paper 2. Aronsson, Markus and Sheeran, Mary. *Hardware software co-design in Haskell.* In: *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell. 2017.*

This paper presents my vision for a hardware software co-design language. I am the lead author.

Paper 3. Aronsson, Markus and Claessen, Koen and Sheeran, Mary and Smallbone, Nicholas. *Safety at speed: in-place array algorithms from pure functional programs by safely re-using storage.* In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing. 2019.*

This paper presents Claessen's original idea for verifying the safety of in-place array updates in functional languages. The writing and development effort was shared equally between myself and Smallbone.

Paper 4 (Draft) Aronsson, Markus and Nordlander, Johan. *Qualified Regions: a system of Qualified Types for inferring regions.*

This paper presents my and Johan Nordlander's vision for a simple region inference system using qualified types. Johan and I developed the idea, I implemented the inference system, constructed the proofs and wrote the paper.

Contents

Part I

Introductory chapters

Chapter 1

Introduction

An *embedded system* is any computer system that is part of a larger system but relies on its own microprocessor. It is embedded to solve a particular task, and often does so under memory and timing constraints with the cheapest hardware that meets its performance requirements. Because performance is such a significant factor in the design of embedded systems, certain systems incorporate more than one kind of processor to benefit from various specialized processing capabilities. Such dedicated processors are often referred to as accelerators, and systems that use various accelerators are in turn referred to as *heterogeneous systems*.

Developing programs for embedded systems requires good knowledge about the architecture that they are supposed to run on; not only should programs be resource efficient, but they should also take full advantage of the hardware to achieve maximum performance. Embedded systems are therefore predominantly developed using a mixture of low-level, imperative languages and hardware description languages, which have an abstraction level that is well suited to extract maximum performance from the various accelerators and memory subsystems.

However, one of the main disadvantages of using low-level languages is that the fine level of control they provide must be exercised at each step in the development of a system; low-level languages force programmers to focus on implementation details rather than functionality or how to best distribute a program over the available accelerators. In particular, the problem of implementing an algorithm often becomes a matter of essentially implementing an algorithm for a specific architecture. Design exploration and code re-use are therefore quite hard to achieve, as the main processor has one and accelerators another, usually very different, architecture.

The difficulty in re-using functionality across different architectures in low-level languages is directly related to a lack of modularity. There are also other issues where the relation to modularity is less obvious, especially for imperative low-level languages. For instance, we would ideally like to treat the partitioning of code over the available processors and accelerators as a modular problem with respect to functionality. While imperative languages can certainly wrap code in run-time conditional statements to switch between local and offloaded implementations, doing so often requires that we first repartition the code into smaller pieces to isolate functionality. Furthermore, such changes lead to extra control logic and can interfere

with optimizations or other design decisions.

I assert that the use of *functional programming* can address many of the issues low-level languages face in the development of embedded systems, because it provides all the necessary tools for abstraction, generalization and modularity that low-level languages sorely lack. It would, however, be naive to suggest that the adoption of functional programming can remedy all ills of embedded systems; one of the fundamental problems with functional programming languages is that it is difficult to give performance guarantees and resource bounds on their programs, a crucial feature for embedded systems. Fortunately, there is a way to gain the productivity and modularity benefits of writing functional programs without incurring the cost of running them: use functional programming to define domain-specific languages for embedded systems.

1.1 Functional systems development

Functional programming is a programming paradigm, a style of programming where the fundamental operation is the application of functions to arguments. A functional program is in fact written as a function that receives input as its argument and returns the output as its result, typically through calling other functions, which themselves are defined by smaller functions still or language primitives. The principal idea behind a functional paradigm is then to treat such a computation as the evaluation of ordinary mathematical functions; it is a declarative programming paradigm that avoids assignment statements, so that variables, once given a value, never change.

One key benefit of functional programming is this focus on describing an algorithm's behaviour through functions and declarations rather than imperative statements, which suffer side effects and have a tendency to get bogged down in implementation details. Functional programs are therefore easier to understand, because they encapsulate state and provides the modularity that enables a programmer to build larger applications by assembling smaller components.

While many different flavours of purely functional languages exist, we have mainly considered Haskell (Peyton Jones et al. 2003). Its most important benefits for embedded software are:

- **Functions are free from side effects**: Without observable side effects, a function's return value is only determined by its input values. This is similar to how a mathematical function works and it means that a function call can be replaced by its final value without changing any other values in the program, which is great for equational reasoning.

- **Functions are higher-order:** The capacity of a higher-order function to take one or more functions as arguments and to return another function as its result means that a program can abstract over entire algorithms, not just values; an expression can be written once and then re-used indefinitely.

- **Evaluation is lazy:** As expressions are not evaluated when they are bound to variables, but rather deferred until their results are needed by other computations, it is possible to avoid needless or intermediate values between most functions. Laziness thus makes it viable to define potentially infinite recursive patterns, which allow for more a straightforward implementation of some streaming algorithms.

Despite these benefits, Haskell and other functional languages are rarely considered in embedded systems development. A major reason for this disregard comes from how difficult it is to predict the time and space costs of evaluating lazy and functional programs (predictable performance is a crucial feature for embedded software because it typically runs under time and memory constraints). Unfortunately, this problem is fundamental to the declarative paradigm of purely functional programming languages, because the amount of work a program needs done, and when it is done, depends on its input values, which are only available at run-time.

As an example, consider the high-level function `map` in Haskell for mapping another function across each element in a list:

```
map _ []     = []
map f (x:xs) = f x : map f xs
```

This recursion is fine in a mathematical setting because the stack (the memory that local values are allocated in) is unlimited. Of course, on real hardware the stack is very much finite and naively traversing a long list of values with recursion can potentially result in a call stack that exceeds the stack bound. Being able to express what a program should do, rather than how to do it, means that a functional program over immutable values cannot match the model of a processor as well as most imperative languages.

In contrast, if we were to implement an example of mapping a function across a collection of values in C instead, then we would perhaps end up with the following:

```
void map(void ** src, void * dest, size_t n, size_t t
        ,void * (f)(void *, void*), void * args)
{
    unsigned int i;
    for(i = 0; i < n; i++)
    {
        void* val = f(src[i], args);
        memcpy(dest, val, t);
        dest = (char*)dest + t;
        free(val);
    }
}
```

We can see from this example that C gives the programmer control over details that were hidden in the Haskell function, such as the choice of data structure and the memory allocation strategy. While the time and space usage is now apparent from simply inspecting the code (provided that the cost model for memory access is straightforward), the programmer is unfortunately also responsible for managing those extra details.

Note that Haskell does include type classes for arbitrary collections, libraries for mutable arrays and does supports a notation close to that of C's imperative programs. But the above example is not meant to compare the potential terseness of functional programs, but rather to illustrates the usefulness of a declarative paradigm for heterogeneous systems: by not explicitly telling the compiler how to do a task, the compiler is free to interpret programs in ways that fit the targeted architecture, whatever that may end up being. For example, the above C implementation is simply one possible interpretation of the mapping function, and we could just as well interpret `map` as a parallel hardware description for some programmable logic units.

In short, the fine level of control that C provides forces developers to make too early design decisions that, once they are made, are difficult to change since they are tightly coupled with each other through code. Functional programming, in contrast, provides abstraction, generalization, and modularity through its higher-order functions, rich type systems, and lazy evaluation, and thus does not exhibit the same issues as low-level imperative languages. The concern with functional programming is instead how one can benefit from its abstractions in a resource-aware setting, where predictable performance and resource usage are important properties.

1.2 Resource awareness in functional languages

One of the more popular solutions to the issue of unpredictable costs in functional programming is to adopt a domain-specific approach, for instance by embedding a special-purpose language within a functional host that is tailored to a certain problem domain. While domain specific languages are typically less comprehensive than general-purpose languages, they are much more expressive in their domain; well-designed domain-specific languages attempt to find a proper balance between expressiveness and efficiency.

One example of a domain-specific language is Feldspar (Axelsson, Claessen, Dèvai, et al. 2010; Axelsson, Claessen, Sheeran, et al. 2011), a functional language that is embedded in Haskell and designed for embedded software development, specifically in the domain of baseband signal processing. Feldspar takes full advantage of its host's functional features to provide a data-centric and modular style of programming with vectors, while simultaneously limiting its own syntax and semantics to give the programmer predictable performance. For example, an idiomatic implementation of a dot product in Feldspar is written in a compositional style with high-level functions from its vector library:

```
dot :: (Type a, Numeric a)
    ⇒ Vec (Data a) → Vec (Data a) → Data a
dot xs ys = sum (zipWith (*) xs ys)
```

In the code above, the type `Data` represents a program fragment in Feldspar; an expression with type `Data Int32` produces a value of type `Int32`. Note that `dot` is polymorphic in its element type, as it accepts any representable type `a` that belongs to its `Numeric` type class. By providing a concrete type for `a`, Feldspar's compiler is able to translate `dot` into efficient C code by exploiting lazy evaluation to eagerly in-line and fuse its vector operations.

The compositional style of a purely functional language like Feldspar not only gives shorter, more succinct definitions than most imperative implementations, but also raises the abstraction level at with the programmer works with algorithms. Indeed, there is no mention of memory allocation or similar low-level details in `dot`. Vectors and higher-order functions are instead used to capture generic patterns or aspects of signal processing algorithms in a manner that is as close as possible to the abstractions used by domain experts reasoning about a problem or solution. It is the restrictiveness of the domain that permits the use of a rather specialised domain specific language and also what makes it possible to achieve the necessary performance.

We mention in particular the Feldspar language because it provided much of the original motivation for this thesis. In fact, our efforts started out as an extension to Feldspar with support for synchronous signal processing (Feldspar previously had a rather low-level interface for dealing with streams and recurrence relations).

Working with Feldspar, however, we found that its purely functional approach meant that programmers lost control of memory allocation and evaluation order, which was left to the Feldspar compiler. For algorithms that relied on a particular memory scheme, the problem manifested itself in both extra memory for storing intermediate results and excessive copying. While Feldspar seeks to solve these issues by embedding controlled side effects in its pure `Data` type, we instead instead took part in the development of RAW-Feldspar (Axelsson et al. 2016), a derivative of Feldspar that complements its data-centric computations with a functional imperative programming paradigm where memory is managed explicitly.

However, as we noted previously, the domain of signal processing is large and the data-centric style of vectors cannot comfortably describe every algorithm, nor can every modern system be implemented wholly in software. For instance, a heterogeneous system's compute elements may have different instruction set architectures or interpretations of memory, both of which may lead to differences in development choices like their preferred programming languages. Could a resource aware Feldspar derivative be of use in such a heterogeneous setting as well? There is certainly some overlap in the programming practices for embedded software and behavioural hardware descriptions, but there are also crucial differences in their programming practices that are necessary to fully exploit a system's capabilities.

1.3 Hardware software co-design with Co-Feldspar

In general, building software systems for embedded heterogeneous systems demands that we generalise functionality and abstract away from particular architectures. For example, design exploration and code re-use are both heavily reliant on platform-independent abstractions, because any intrinsic operations of one architecture may not be available on another. On the other hand, performance relies on the ability to tailor designs and configurations to a particular environment, specifically to make use of intrinsic or accelerated operations.

While the desire for performance seems to go against the above desire for modular and generic programs, we note that such optimisations should primarily be considered late in development; a strong focus on performance during the initial development of a system is a kind of premature optimisation (Persson 2014). It is premature partly because it forces developers to make early decisions that, once they are made, are tightly coupled through the implementation and therefore difficult to change. After all, a focus on low-level details throughout the entire development process is one of the primary reasons for the low re-usability and modularity of algorithms written in languages like C.

We would therefore argue that a functional domain specific language like Co-Feldspar gives a compelling approach to developing embedded heterogeneous systems: start with a modular, generic, functional algorithm; specialise its components as needed for the targeted architecture. For it is functional features like higher-order functions, lazy evaluation and a rich type system that enable programmers to build applications by assembling smaller functions, while the restrictiveness of the domain gives us hope of achieving the necessary performance. To be more specific, it is the following features that we consider essential for functional language that target embedded heterogeneous systems, and aim to explore with Co-Feldspar:

- **Flexible interpretation and design**: The presence of multiple processing elements in general means that we cannot make broad assumptions about the systems our programs will run on; Co-Feldspar must be modular in the perspective of both its users and implementers.

- **Intuitive behaviour**: Both the syntax and semantics of Co-Feldspar should accurately capture signal and vector algorithms in its problem domain and reflect their performance characteristics; an application programmer should be able to predict the resource bounds of their designs. We focus on memory management in particular.

- **Efficient and safe abstractions**: Because predictable performance is such a crucial feature for embedded systems, any extension of Co-Feldspar must also be accompanied by an efficient elaboration into its simpler core language. In general, it is not acceptable for a compiler to generate, for instance, excessive copying and memory for intermediate values.

Paper 2 gives a detailed introduction of the Co-Feldspar[1] language, so named because of its focus on co-design and predecessor Feldspar. Co-Feldspar is a staged language, it is embedded in Haskell and capable of generating both C and VHDL code, including code for connections between software and hardware components. The staged approach makes it possible to postpone many of the implementation decisions to later stages of the development process, and provides a reasonable degree of control over the generated code.

Like Feldspar, we make extensive use of functional features in Co-Feldspar to build combinator libraries, but we also make sure to abstract away software specific types and operations. For example, the previous dot product can be re-implemented in Co-Feldspar as:

```
dot :: (Type exp a, Vector exp, Num (exp a))
    ⇒ Vec exp a → Vec exp a → exp a
dot xs ys = sum (zipWith (*) xs ys)
```

Here, the interpretation of `dot` is constrained to any expression type `exp` that supports vectors, rather than a software or hardware specific `Data` type; `dot` only contains generic vector functions and can be compiled to both C and VHDL. In contrast to Feldspar, however, the input vectors of `dot` must be explicitly declared by the programmer through a monadic interface:

```
prog = do a ← initArr [1..5]
          b ← initArr [5..9]
          return (dot a b)
```

The generic `dot` enjoys the same functional benefits as its implementation in Feldspar did, but the programmer now has full control over its memory use and evaluation order. In a sense, the programmer also has control over the possible interpretation of `dot` because type classes like `Vector` can be used to guarantee presence, and absence, of functionality in `exp`. Software and Hardware specific types and operations, and interfaces between the two, are available through similar type classes or once `exp` has been instantiated. That is, Co-Feldspar enables programmers to express the entire design process for applications on heterogeneous architectures with embedded software and hardware components, including the necessary exploration to decide where the boundary between components should be.

1.3.1 Synchronous data-flow

Signal processing is however more than just sequencing computations, it is also about how to connect those computations in a network that operates on streaming data. However, such signal networks are typically expressed as data-flow graphs over pure stream transformers and where state is introduced explicitly through unit delays. A question, then, is how one can efficiently express such signals on top of a monadic

[1] Available as open-source at: github.com/markus-git/co-feldspar.

language like Co-Feldspar while still permitting a functional style of programming with pure functions operating on immutable signals.

Paper 1 explores such a signal processing library[2], which extends an existing domain-specific language with support for synchronous data-flow programming. Practically, the library provides a means to connect expressions in the underlying language using a model of synchronous data-flow networks. The signal compiler then reifies such networks into an imperative representation of the classical one for co-iterative streams (Aronsson 2014; Axelsson, Persson, and Svenningsson 2014), simplifying the compilation of signals into imperative programs.

From the user's perspective, the signal library supports definitions in a traditional functional programming style with high-level combinators, reducing the gap between the mathematical description of signal processing algorithms and their implementation. This combinatorial style of programming, combined with support for unit delays and recursively defined signals, provides a simple but powerful syntax that allows programmers to express any kind of logic networks with memory and feedback loops.

1.3.2 Virtual array copies

While monadic languages like Co-Feldspar can express in-place updates through mutable arrays, Haskell's type system is not strong enough to guarantee that doing so is safe; it is easy to mistakenly overwrite some input data too early, especially when computations are written in an imperative style. Such mutable updates are not visible at the type level in Haskell, and its type system therefore provides no help in avoiding such mistakes. Nevertheless, in-place updates are an essential part of certain algorithms and we would like to provide some safety for the programmer.

Paper 3 explores an array programming library[3] with support for virtual array copies. A virtual copy of an array gives the illusion of being a real copy of the array, and semantically it behaves as if it really were a copy. No such copy is however made. Rather, the virtual copy makes an alias of the original array, a second pointer to the same underlying values. The array library then enlists the help of a theorem prover to verify that the illusion of any virtual copy is preserved in a whole, closed program. Because the compiler checks that executing selected computations in-place does not change their original semantics, users can freely use equational reasoning to understand or implement their algorithms.

1.4 Automatic but predictable memory inference

Monadic interfaces like that of Co-Feldspar have been successfully applied to model statefull programming in a number of domain-specific languages. One reason for their success is perhaps that they offer the only really satisfactory solution to imperative functional programming in Haskell (Peyton Jones and Wadler 1993). The monadic

[2]Available as open-source at: github.com/markus-git/signals.

[3]Implemented as part of Co-Feldspar.

interface is however quite pervasive, and it can get in the way of implementations that do not depend on a particular evaluation order or memory use.

As an example, consider the following program stub:

```
do x ← return (1 + 2)
   y ← return (3 + 4)
   return (x + y)
```

Here the ordering of the two statements is not the criterion for their evaluation; although monads enforce a particular ordering of the two expressions, they could really be evaluated in any order, even in parallel. Furthermore, a monadic program is essentially a sequence of statements with pieces of pure expressions scattered across each statement. These expressions are typically quite small, so high-level optimizations are only performed for small parts of a program.

There are thus good reasons for programmers to every now and then prefer a more traditional style of functional programming where memory is automatically managed. Ideally, only when the compilation results in inefficient memory usage would there be a need for programmers to step in and address the problem manually.

An alternative approach to monadic encapsulation of memory effects is the use of a *type-and-effect* system (Talpin and Jouvelot 1994). These systems infer not only the type of an expression, but also the computation effects that may happen during its evaluation. Most interesting is perhaps that, while monads demand that users manually merge values with computational descriptions, type-and-effect systems typically require little to no user interaction or modification of their programs; they automatically infer a safe approximation of the memory use in a program (Talpin and Jouvelot 1992).

One of the more influential ideas to come out of type-and-effect systems is the inference of regions (unbounded areas of memory intended to hold heap-allocated values). Tofte and Talpin (1997; 1994) showed how a type-and-effect system could be used to automatically assign a stack-like memory discipline to strict functional programs. The memory store in this discipline is essentially a stack of regions, where each region grows in size until it is removed in its entirety. A program analysis then automatically identifies points at which entire regions can be allocated and de-allocated, and it decides into which region each value should be put. All values, including function closures, are put into regions.

1.4.1 Region inference based on qualified types

While regions provide a compelling approach to memory management, the type-and-effect systems behind their inference are unfortunately quite complicated. Much of this complexity comes from how difficult it is to perform an accurate lifetime analysis for values in higher-order functions, particularly in those that call themselves recursively (Tofte, Birkedal, Elsman, and Hallenberg 2004). In fact, most region-based languages struggle with tail recursion and iteration in general.

Fortunately, most languages that target embedded systems already refrain from using unbounded recursion, because it is difficult to predict the time and space costs of its evaluation. A functional language designed to be suitable for implementation of embedded software can thus benefit from a simplified type-and-effect system that avoids the difficulties of recursion while still permitting us to express interesting programs. We explore such a restricted system for region inference, where the inference rules are based on a notion of *qualified types* (Jones 2003) rather than effect tracking.

The main benefits of the new system for qualified regions are:

- **Tailored to embedded systems**. By considering functional languages without recursion, the region labelling of expressions can be simplified for an important subset of languages.
- **Functions without thunks**. Without general recursion, thunks do not have to be allocated. This simplification, combined with an approximation for when a region can be de-referenced by a function, considerably limits the need for tracking effects in types.
- **Syntax-directed inference**. We extend the type inference algorithm W (Milner 1978) such that a typing is an entailment from a term to its region annotated form, where the context includes a set of allocated regions.

1.5 Research questions

We have argued that low-level imperative languages force implementers to focus on non-functional details rather than functionality and how to best utilize the available architecture. Furthermore, any decisions for such details will inevitably end up tying their implementations to a particular system, leading to low re-use and low modularity. The heterogeneity of modern embedded systems further aggravates these issues, as processors and accelerators usually have very different architectures.

We assert that embedded, heterogeneous system development can benefit from ideas in functional programming, because the functional paradigm provides the necessary abstractions and modularity to build applications by assembling re-usable and generic components. To narrow down our exploration, we have focussed our efforts to a modern FPGA with embedded ARM cores. We consider in particular the generation of behavioural VHDL descriptions for its programmable logic and C for one of its embedded cores. However, if given an extensible model of hardware and software, the step to support other accelerators with similar architectures should be relatively small. To summarize, we formulate the following research questions:

1. Can the embedded, heterogeneous system development for modern FPGAs benefit from a functional hardware software co-design language? In particular:

   (a) Can a low-level, core language be embedded within a functional host to provide a model of the various imperative languages used to describe heterogeneous systems that is both extensible and has a relatively small semantic gap to C and VHDL?

   (b) Can such an embedded core language exploit the higher-order functions and rich type systems of its host to separate the syntax of its programs from their semantics such that a program can be parameterised by the functionality it requires and, once written, can be interpreted in a variety of ways?

2. How can imperative functional programming be extended with support for efficient, synchronous data-flow definitions, reducing the gap between streaming algorithms and their implementation in an otherwise imperative language?

3. Can a functional array programming language offer safe, in-place array transformations without neither weakening its transparency nor the ability to apply equational reasoning?

4. Does a functional language without recursion permit the definition of a region-based memory management scheme that is simpler than standard type-and-effect systems for region inference?

The exploratory research questions regarding embedded languages, stream processing and in-place updates are investigated by building the Co-Feldspar langauge, a derivative of Feldspar for hardware software co-design with explicit memory management, and the signal and array programming libraries. More specifically, we will build upon our core language on a deeply embedded model of imperative programs as monads (Svenningsson and Svensson 2013; Apfelmus 2016), and employ a technique similar to data types à la carte (Swierstra 2008; Axelsson 2019) to support extensible definitions and interpretations. Practically, we aim to expand previous work in RAW-Feldspar and its deep embedding of software programs (Aronsson, Axelsson, et al. 2019) with support for hardware software co-design. We will then exploit Haskell's higher-order functions and rich type system to build shallowly embedded combinator libraries on top of the monadic core that permit a more functional programming style. The signal and array programming libraries will be built in a similar fashion, but instead explore the use of co-iterative streams and theorem provers in their respective domains.

Seeing as their definition is primarily a practical result, evaluation is carried out by a variety of tests and benchmarks of real-world examples. On the other hand, our work with the simplified system for region inference is mostly theoretical. We therefore argue for the usefulness of our region system by proving that it is correct and that its region labelling preserves the source program's original meaning.

1.6 Review and organisation of thesis

Chapter 1 and its subsections try to highlight the many interlocking issues in developing embedded software and hardware code for heterogeneous systems with low-level languages. In particular, the use of low-level languages makes it next to impossible to write code that is re-usable, modular and high-performance at the same time. Section 1.1 argues that functional programming languages provide the necessary modularity and abstraction build applications by composing small and re-usable functions, but instead suffer from unpredictable performance.

In response to these challenges, we are developing Co-Feldspar (Section 2.2 and Paper 2), a functional hardware software co-design language with explicit memory management. To complement the imperative functional programming paradigm that Co-Feldspar is built on, we have developed libraries of high-level combinators for signal processing (Section 2.4 and Paper 1) and safe, in-place array updates (Section 2.5 and Paper 3). Finally, we have explored a memory management scheme that is based on regions and specialised for embedded software (Section 2.6 and Paper 4) as an alternative to the explicit memory management in Co-Feldspar.

Chapter 2 and its subsections introduce the design decisions behind Co-Feldspar, the signal and array programming libraries, and the region inference system in more detail and showcases some use cases through signal processing examples. In particular, Section 2.2 introduces the core of Co-Feldspar, and Section 3 its design decisions. Section 2.4 and 2.5 introduces the design decisions behind the signal and array programming libraries together with various use cases. Section 2.6 gives the intuition behind our region inference system.

Chapter 2

The Co-Feldspar language

Chapter 1 introduced heterogeneous computing as an interesting development in the domain of embedded systems, but noted that its development is not without its own challenges. As a more concrete example of these challenges, consider a modern FPGA (Chung et al. 2010), a system that shows great promise as a prototypical heterogeneous system with configurable computing capabilities. Despite their many benefits, however, the adoption of FPGAs has been slowed by the fact that they are difficult to program efficiently (Teich 2012).

The logic blocks of an FPGA are usually programmed in a hardware description language like VHDL or Verilog, while its embedded processors and co-processors are typically programmed in some low-level dialect of C or even assembler. This choice of languages is motivated by a desire to extract maximum performance from the FPGA's hardware and software components, and these languages have constructs that are well suited to fine-grained control over such components.

As mentioned previously, this control come at a cost—a programmer cannot abstract away from the specific system architecture, but must keep the implementation on a low level during the entire design process. Combined with the fact that programs are often heavily optimized to fit their constraints, this means programmers inadvertently end up tying their code to whatever component it is running on. However, our modern FPGA gained its performance by not just adding more processors of the same kind, but by incorporating co-processors with specialized architectures and exploiting its programmable logic to handle particular tasks. Low-level languages provide little support for the design exploration necessary to find a good partitioning of code on such systems.

Many of the aforementioned issues with low-level languages are related to lack of modularity. Some are directly related, such as the architectural issues, and others are indirectly related. For example, issues like parallelism would ideally be treated as a modular and separate concept with respect to functionality. It is of course possible to code for different architectures into a single function and use conditional statements to switch between implementations. But it is often necessary to repartition code into smaller conceptual pieces to exploit accelerators, which then leads to extra control paths that can get in the way of efficient compilation.

In his seminal paper "Why functional programming matters" (Hughes 1989),

Hughes argues that many of the problems with low-level languages can be addressed using concepts from functional programming. In particular, the glue code that functional programming languages offer, through higher-order functions and lazy evaluation, greatly increases ones ability to modularise a problem conceptually. The benefits of functional programming are, however, not limited to describing software, as Sheeran argues in her paper "Hardware Design and Functional Programming: a Perfect Match" (Sheeran 2005). Sheeran exemplifies how a functional language can make it easy to explore and analyse hardware designs in a way that would have been difficult, if not impossible, in traditional hardware description languages.

The question, then, is how to benefit from functional concepts in heterogeneous systems, where predictable performance is a crucial feature and functional languages are rarely considered. Indeed, the lack of predictable memory use and performance is a fundamental problem to the declarative paradigm behind functional programming; when pure functions are defined by composing high-level combinators, the programmer loses control over "how" and "when" their code is executed.

Fortunately, there is a way of retaining the productivity benefits of writing programs in a functional style without incurring the cost of running them: use functional programming languages to design and embed domain-specific languages. Such languages can not only enjoy the abstractions and modularity of their functional host, but can also limit themselves to features that can be represented efficiently as source code in its targeted domain (Hudak et al. 1996). The restricted domain permits the use of a rather specialised language, while also giving hope in achieving the necessary performance.

2.1 Benefits of a domain-specific approach

A domain-specific language is a special-purpose language, tailored to a certain problem and captures the concepts and operations in its domain (Hudak et al. 1996). Domain-specific languages represent a popular means to improve productivity for a certain domain of problems (Fowler 2010). For instance, a hardware designer might write in VHDL, while a web-designer who wants to create an interactive web-page would use JavaScript. The general idea is that the constructs of domain-specific languages should be as close as possible to the concepts used by domain experts when reasoning about a problem or its solution.

Domain-specific languages come in two fundamentally different forms: external and internal, where VHDL and JavaScript are both examples of the former. Internal languages, on the other hand, are embedded in a host language, and are often referred to as embedded domain-specific languages. The advantage of the embedded approach is that a domain-specific language inherits the "look and feel" of its host language, and its generic features such as modules, type classes, abstract data types and higher-order functions. Haskell, with its static type system, flexible overloading and lazy semantics, has come to host a range of embedded languages (C. Elliott et al. 2003). For instance, popular libraries for parsing (Leijen and Meijer 2002), pretty printing (Hughes 1995), and hardware design (Bjesse et al. 1998; Gill et al. 2010; Bachrach et al. 2012a; Baaij et al. 2010) have all been embedded in Haskell.

Embedded languages in Haskell are typically further divided into a shallow or deep style of embedding. Conceptually, a shallow embedding captures the semantics of the data in a domain, whereas a deep embedding captures the semantics of the operations in a domain. Both kinds of embeddings have their own benefits and drawbacks. For example, it is easy to add new functionality to shallowly-embedded types, whereas a deeply embedded type is static but facilitates interpretation in different domains. While the implementations of deep and shallow embeddings are usually at odds, there has been work done to combine their benefits (Svenningsson and Axelsson 2012). A mixture of shallow and deep embeddings ensures that the core is easy to interpret while simultaneously allowing user facing libraries to provide a terse and extensible syntax (Axelsson, Claessen, Sheeran, et al. 2011).

2.2 Embedding Co-Feldspar

Co-Feldspar, our hardware software co-design language, is implemented as an embedded language in Haskell with a mixture of shallow and deep embedding techniques, which means that primitive language constructs are provided as ordinary Haskell functions. These functions do not perform any actual computation, but instead build data structures that represent the corresponding imperative program that makes out the core of Co-Feldspar. A variety of shallow embeddings then complement this core with combinator libraries. For example, a hierarchy of type classes for common and language specific types and operations is used to provide an extensible and structured way of controlling the overloading in programs.

In general, the main characteristics of Co-Feldspar's embedding are:

- **Imperative functional programming**: Imperative programs are shallowly embedded into Co-Feldspar through its monadic interface; a program inherits the scope of its monadic generator, and if the generator is well-typed, then so is the embedded program. Furthermore, the monadic interface makes it possible to describe algorithms that, for performance, rely on destructive updates, or on a specific evaluation order or access pattern.

- **Overloaded instructions**: Programs in Co-Feldspar with purely computational instructions are distinguished only by types. The boundary between software and hardware for generic programs can be moved simply by instantiation; no additional syntactic annotations are required.

- **Extensible interpretation and data-types**: Co-Feldspar's model of imperative programs is parameterised by its instructions, expressions and type predicates, which means that hardware and software programs share a common core. This polymorphism, combined with an extensible core, means that different interpretations and extensions can be introduced separately.

A user interface structured with the help of Haskell's type classes has two big advantages: it is a powerful modelling tool, because it gives a concise and precise description of languages, and it helps in factoring out shared operations and

abstractions between languages. For instance, Co-Feldspar defines a type class that captures the fact that both C and VHDL support a similar notion of variables:

```
class Monad m ⇒ References m where
  type Ref m :: * → *
  newRef  :: SyntaxM m a ⇒ m (Ref m a)
  getRef  :: SyntaxM m a ⇒ Ref m a → m a
  setRef  :: SyntaxM m a ⇒ Ref m a → a → m ()
```

This class, which we named `References`, introduces shared instructions and an abstraction `Ref` for variables in the monadic program type `m`. Further, note that each instruction constrains its element type `a` by `SyntaxM m`, as it ensures that `a` is representable in `m`.

Practically, a type class like `References` provides a means to reason about the presence, and absence, of certain instructions in a program. For example, the following function is only allowed to use variable instructions:

```
updateRef :: (Monad m, References m, SyntaxM m a)
  ⇒ Ref m a → (a → a) → m ()
updateRef r f = do v ← getRef r
                   setRef r (f v)
```

Not all instructions need to be part of the core language. In fact, one benefit of a deeply embedded language is the ability to use the host language to generate programs. This allows us to define type classes that provide complex language constructs as generators that translate into more primitive constructs. For example, consider Co-Feldspar's type class for let-bindings:

```
class Share exp where
  share :: (Syntax exp a, Syntax exp b)
        ⇒ a → (a → b) → b
```

`share` accepts any type `a` and `b` that are internally represented as expressions of type `exp` (`Syntax` is a non-monadic version of `SyntaxM`), and allows us to, for instance, avoid duplicating a `heavy` computation:

```
fun :: (Monad m, Share (Exp m), SyntaxM m Int32)
  ⇒ Exp m Int32 → Exp m Int32
fun ref = share (heavy) (λx → x + x)
```

where `Exp` is a type family that gives us the expression type associated with `m`. While we cannot translate `fun` directly to either C or VHDL, as neither language has a primitive construct for let-bindings, we can elaborate `share` into a program stub with references:

```
fun :: (Monad m, References m, SyntaxM m Int32)
  ⇒ Ref m Int32 → m (Exp m Int32)
fun ref = do r ← initRef (heavy)
             v ← getRef r
             return (v + v)
```

Translation `fun` into C and VHDL is now straightforward. In fact, Co-Feldspar's compiler is expressed as a series of such program transformations, turning high-level constructs into equivalent, but still efficient, program stubs with simpler instructions. In this sense, our approach to code generation is similar to previously published methods (Sculthorpe et al. 2013; Svenningsson and Svensson 2013; Axelsson 2016), and the design of Co-Feldspar is reminiscent of the light weight modular staging in Scala (Rompf and Odersky 2015; George et al. 2013b) or the Habit project (Jones 2013) and its intermediate language MIL (Jones et al. 2018).

2.3 Programming with Co-Feldspar

Programming in a monadic language like Co-Feldspar is similar to programming in an imperative language like C but also not quite the same. As an example of the differences between the two styles, and to showcase the co-design language, consider a finite impulse response (FIR) filter, one of the two primary types of digital filters used in digital signal processing applications (Oppenheim et al. 1989). A FIR filter is, in short, a filter whose impulse response settles to zero in finite time. For a causal discrete-time FIR filter of rank N, each value of the output sequence is a weighted sum of the $N+1$ most recent input values and the filter is typically defined as follows:

$$y_n = b_0 x_n + b_1 x_{n-1} + \cdots + b_N x_{n-N} = \sum_{i=0}^{N} b_i x_{n-i}$$

where x and y are the input and output signals, respectively, and b_i is the value of the impulse response at time instant i (and an N'th-rank filter has $N+1$ terms on the right-hand side). The FIR filter can be implemented in C as:

```
void fir(
  double *x, size_t L, double *b, size_t N, double *y)
{
  size_t n, j, min, max;
  for (n = 0; n < L+N-1; n++)
  {
    min  = (n >= N-1) ? n-(N-1) : 0;
    max  = (n+1 < L) ? n+1 : L;
    y[n] = 0;
    for (j = min; j < max; j++) y[n] += x[j] * b[n-j];
  }
}
```

where L is the length of the input, N is the filter rank, and b, x, and y are pointers to the filter's coefficients, input, and output, respectively.

At first glance, the C code seems to be a good representation of the FIR filter, but there are a few modularity problems with its implementation. For instance, the inner for-loop calculates a shifted dot product of the arrays b and x inline, a fairly common operation in signal processing. We would like to define it once and then re-use whenever needed. It is of course possible to move the operation to a stand alone function:

```
double dot(
  size_t min, size_t max, size_t n, double *x, double *b)
{
    size_t j;
    double sum = 0;
    for (j = min; j < max; j++) sum += x[j] * b[n-j];
    return sum;
}
```

However, the function is restricted to values of type *double*, it assumes that b and x both have elements in the range between min and max, and it is not compositional in the sense that it cannot be merged with the producers of b and x without looking at their implementation.

The same shifted dot product can be implemented in Co-Feldspar as a software expression using a similar, although not idiomatic, style:

```
dot :: SExp  Length → SExp  Length → SExp Index
    → SIArr Float   → SIArr Float  → SExp Float
dot min max n x b =
  loop min max 0 (λj s → s + x!j * b!(n-j))
```

`SIArr` and `SExp` are software types and represent immutable arrays and expressions,

respectively. In general, we use an `S` prefix for software types and `H` for hardware types. The iteration scheme used to compute the dot product is captured by `loop`, a high-level combinator with the following type signature when instantiated to software expressions:

```
loop :: SExp Length → SExp Length → SExp a
     → (SExp Index → SExp a → SExp a) → SExp a
```

The first and second parameters of `loop` are the iteration range, the third is the initial loop state and the fourth parameter is the iteration step function, which calculates a new state from the current loop index and the previous state.

The above implementation is not without its own faults. We can improve it by, for instance, making it polymorphic in its element type and thus able to accept more types than just `Double`. But more importantly, its implementation is also limited by its use of the software specific types and operations, because hardware languages also support the iteration, arrays and numerical operations used by `dot`. We can therefore improve the function even further by replacing its types and operations with generic ones. Actually, every operation in `dot` already comes from one of Co-Feldspar's type classes and we have simply instantatied them to software. So we need only update its type signature to turn `dot` into a generic program:

```
dot :: (Common exp a, Finite exp arr)
    → exp Length → exp Length → exp Index
    ⇒ arr a      → arr a      → exp a
dot min max n x b =
  loop min max 0 (λj sum → sum + x!j * b!(n-j))
```

Here, `Finite` ensures `arr` supports indexing, and `Common` is a short-hand for expressions commonly found in both software and hardware, like `loop`:

```
type Common exp a = (Iterate exp, Num (exp a), ...)
```

The final `dot` can be interpreted as both software and hardware by simply instantiating its monadic type `m` as the program type in either language. That is, the use of type classes enabled the separation of an operation's interface from its implementation, which gives Co-Feldspar some freedom when interpreting the meaning of such functions.

2.3.1 Offloading computations to hardware

The more interesting use of Co-Feldspar is perhaps when we offload a generic function like `dot` to hardware and then call it from a FIR filter in software. Thanks to the previous generalisation, interpreting `dot` as a hardware function is straightforward: simply instantiate it as a hardware program. But to reach a hardware program from software one must first set up a communication channel to that program.

An isolated hardware `dot` is, however, of little use on its own; we must also give it an interface that allows software programs to cooperate with it. The construction of such interfaces is typically guided by a set of rules that, in our co-design example, would translate a single hardware type into a single software type. Such a simple approach can however be quite limiting as, for instance, a signal of bits is typically used in hardware to represent a variety of values. For this reason, we employ a small language of programmable signatures (Axelsson and Persson 2015) that lets the programmer describe the mapping of each argument.

This little language of mappings is mostly an assortment of functions for requesting various types of inputs and a few return statements to finalize the signature with its output. For example, we can define a straightforward mapping of arguments to `dot` as follows:

```
dotC :: HSig (
     Signal Length → Signal Length → Signal Index
  → SArr Int32     → SArr Int32     → Signal Int32
  → ())
dotC =
  input $ λmin → input $ λmax → input $ λn →
  inputIArr 20 $ λx → inputIArr 20 $ λb →
  retExpr $ dot min max n x b
```

where `input` asks for a scalar value, `SArr` for an array of some known length, which we have set to 20, and `retExpr` finalises the signature by returning a pure expression. Note that `()` marks the end of the signature, and that we have swapped the floating point numbers for fixed point arithmetic to simplify the hardware interface.

Such a straightforward mapping is however not the only choice we have at our disposal, because signatures are wrappers to hardware programs and therefore give access to both generic and hardware specific instructions. For instance, to ensure that `dotC` can be synthesised, we could introduce a few instructions into the interface that limit `dot` to a static interval, slicing the input arrays to their interesting ranges. Such a modification leaves the signature and inputs intact; only `retExpr` and its body needs to been updated on account of the additional instructions:

```
dotC = ... $ ret $ do
  x' ← initArr (replicate 20 0)
  y' ← initArr (replicate 20 0)
  copyArr (x', min) (x, min) (max-min)
  copyArr (b', min) (b, min) (max-min)
  return $ dot 0 20 n x' b'
```

where `copyArr` copies a slice of one array to another, and `ret` finalises the signature with a hardware program. The interface ends up in the generated hardware as a wrapper for `dot`.

A signature like `dotC` can already be used to describe interface by which hardware components communicate. However, before we can call `docC` from software, we must connect its signature to an interconnect. For this reason, Co-Feldspar provides an `axi_lite` combinator that implements an AXI4-lite interconnect, a channel that provides a simple, low-throughput memory-mapped communication between hardware and software. The type signature of this combinator is given in Figure 2.1, and compiles to a hardware component with a similar interface that should immediately be recognised as an AXI4-lite interconnect by a synthesiser like Vivado (Feist 2012). Further, with the help of a synthesis tool like Vivado, we can turn these wrapped designs into physical ones and offload them to hardware. In our case, that piece of hardware is the programmable logic on a Xilinx Zynq (Xilinx 2018).

It is through the physical address of an offloaded design and memory mapped I/O that software programs in Co-Feldspar are finally able to call a hardware component. As we run our examples on a processor with a variant of Linux installed, communication between hardware and software is done through `mmap`: a function that maps kernel address space to a user address. Co-Feldspar implements its own `mmap` function, which wraps the one Linux provides and computes the addresses of each input and output in a signature. Assuming we have offloaded `dot` at an address of "0x83C00000", we can set up a software interface that calls `dot` as follows:

```
dotS :: SRef Length → SRef Length → SExp Index
  → SArray Int32 → SArray Int32
  → Software (SExp Int32)
dotS min max n x b = do
  dot ← mmap "0x83C00000" dotC
  res ← newRef
  nr  ← initRef n
  call dot (min >: max >: nr >: x >>: b >>: res >: nil)
  getRef res
```

where `(>>:)`, `(>:)` and `nil` are used to construct a list of software arguments that matches the signature of `dotC`.

`dotS` is a software program like any other, and could be used in place of the generic `dot` in a software implementation of a FIR filter. To give an example of this, we first define a mostly imperative FIR filter in Co-Feldspar using nonidiomatic code:

```
fir :: (Comp m, Common (Expr m))
  ⇒ IArr m Int32 → IArr m Int32 → m (Arr m Int32)
fir x b = do
  let xl = length x
      bl = length b
  ys  ← newArr (xl+bl-1)
  for 0 1 (xl+bl-1) $ λn → do
    min ← shareM (n ≥ xl-1 ? n-xl-1 $ 0)
    max ← shareM (n+1 < bl ? n+1 $ bl)
    setArr y n (dot min max n x b)
  return y
```

where `shareM` is a monadic version of `share`, and `Comp` is a short-hand for purely computational instructions:

```
type Comp m = (Monad m, References m, Share (Exp m), ...)
```

Nevertheless, given such an implementation of the filter, we need only instantiate `m` as the software monad and swap out the generic `dot` for its offloaded variant `dotS`:

```
fir :: SIArr Int32 → SIArr Int32 → S (SArr Int32)
fir x b = do ...
  for - 1 (xl+bl-1) $ λn → do
    min ← initRef (n ≥ xl-1 ? n-xl-1 $ 0)
    max ← initRef (n+1 < bl ? n+1 $ bl)
    v   ← dotS min max n x b
    setArr y n v
```

As you might imagine, running the FIR filter with an offloaded `dot` does not yield particularly good performance; the updated filter in fact performs worse than running `fir` solely in software. Figure 2.2 shows the avrage execution time for various hardware software partitionings of the FIR filter. One source of inefficiencies is certainly the direct mapping of inputs to arguments in `dotS`, because it ends up sending over the entire arrays x and b each time `dotS` is called, only to process a small part of them in all but one call. While we could perhaps improve `dotC` to operate on array segments, the ratio between computations and communication would be skewed regardless. A better solution is simply to offload the entire `fir` filter.

To offload the FIR filter, we can either re-instantiate its program type `m` as the hardware monad and then substitute `dotS` for a port-map to `dotC`, or simply revert to its generic implementation. Regardless of the approach we take, we will need to create a new signature for the filter:

```
axi_lite :: HSig sig → HSig (
      Signal (Bits 32) -- Write address
   → Signal (Bits 3)   -- Write protection type
   → Signal Bit        -- Write address valid
   → Signal Bit        -- Write address ready
   → Signal (Bits 32) -- Write data
   → Signal (Bits 4)   -- Write strobes
   → Signal Bit        -- Write valid
   → Signal Bit        -- Write ready
   → Signal (Bits 2)   -- Write response
   → Signal Bit        -- Write response valid
   → Signal Bit        -- Response ready
   → Signal (Bits 32) -- Read address
   → Signal (Bits 3)   -- Protection type
   → Signal Bit        -- Read address valid
   → Signal Bit        -- Read address ready
   → Signal (Bits 32) -- Read data
   → Signal (Bits 2)   -- Read response
   → Signal Bit        -- Read valid
   → Signal Bit        -- Read ready
   → ()
)
```

Figure 2.1: Type signature of the AXI4-lite wrapper.

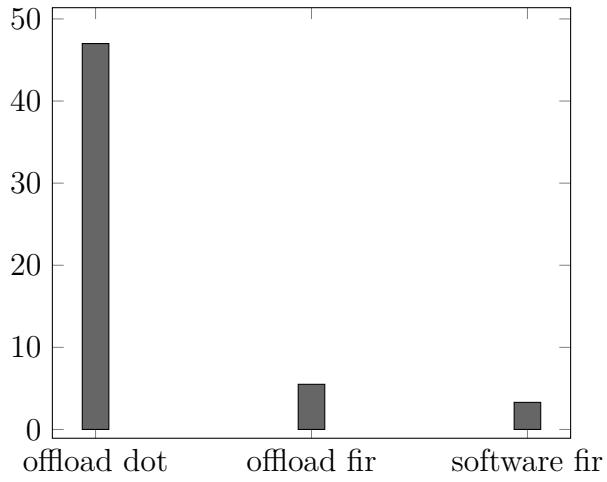

Figure 2.2: Execution time of offloaded functions in microseconds.

```
firC :: HSig (
     SArr Int32 → SArr Int32 → SArr Int32 → ())
firC = inputIArr 20 $ λx →
       inputIArr 20 $ λb →
       retVArr   39 $ fir x b
```

Even though the filter is quite small, it performs roughly as well as one running entirely in software.

This example shows the general design philosophy behind our approach to co-design. Start with a clear, generic implementation of an algorithm and then establish a hardware software partitioning by setting up the required interfaces. Language specific operations and optimisation can then be introduced once a satisfactory partitioning has been found. With this approach, the amount of code that has to be rewritten during initial exploration is limited to the hardware software interfaces. Note that these interfaces are the aforementioned programmable signatures or some other type-guided translation and not the full interconnect; Co-Feldspar is capable of automatically generating the glue code that allows hardware and software components to communicate.

2.3.2 Data-centric vector computations

The sequential implementations of `dot` and `fir` in Section 2.3 are however not idiomatic Co-Feldspar and the code is quite fragile. For instance, the manual indexing of the arrays `x` and `b` is a source of concern that cannot be addressed in the sequential approach; what if the programmer accidentally indexed `x` twice? The program would still type check, but it would not behave correctly. Furthermore, the caller has to assert that both arrays are of the same length. Also, `dot` is not compositional, because it cannot merge with the producers of `x` and `b` without creating any intermediate arrays.

In order to support a higher-order and compositional style of array programming—with less opportunity to shoot oneself in the foot—Co-Feldspar provides an implementation of *pull* arrays, an abstraction built on top of the mutable arrays in Co-Feldspar. As their name implies, pull arrays excel at pulling out values from an array and provide a rich set of combinators and functions to work with flat, or possibly nested, arrays. Implementation wise, a pull array consists of a length and a function from indices to values:

```
data Vec exp a where
  Pull :: exp Length → (exp Index → a) → Vec exp a
```

Pull arrays are notable for their non-recursive definition, which enables aggressive fusion; the composition of two pull arrays will not allocate any intermediate memory at run-time. As an example, consider the following vector functions that we have instantiated to software:

```
zipWith :: (a → b → c) → SVec a → SVec b → SVec c
zipWith f xs ys = Pull (length xs `min` length ys)
  (λi → f (xs!i) (ys!i))

fold :: (a → a → a) → a → SVec a → a
fold f init xs = iter (length xs) init
  (λi st → f (xs!i) st)
```

`zipWith` constructs a new pull array by "pulling" values from both `xs` and `ys` at the same time, and `fold` is defined in terms of `iter`.

We introduced these three functions in particular because we can use them to define a dot product in more idiomatic Co-Feldspar code. That is, a dot product is calculated by first multiplying the vectors `xs` and `ys` element-wise using `zipWith`, and then reducing the result with `fold (+)`:

```
dot :: Num (SExp a) ⇒ SVec a → SVec a → SExp a
dot xs ys = fold (+) 0 (zipWith (*) xs ys)
```

This definition is certainly terser and closer to the original mathematical specification than the sequential one. It is also easier to reason about and sturdier, in the sense that many of the implementation details a user could get wrong (such as indices and lengths) are now hidden. Furthermore, the composition of `zipWith` and `fold` results in a function that, once evaluated, returns a single array where all intermediate values have been eliminated:

```
dot xs@(Pull lx fx) ys@(Pull ly fy)
  = fold (+) 0 $ zipWith (*) xs ys
  = fold (+) 0 $ Pull (lx `min` ly) (λi → fx i * fy i)
  = iter (lx `min` ly) 0 (λi st → (fx i * fy i) + st)
```

That is, by the time the compiler is called, pull arrays are all but evaluated away and only leave behind low-level programs with optimized loops.

Vectors can comfortably describe the regular array transformation that is a dot product, but they can also describe some of the more irregular transformations, like the recurrence relation of a FIR filter. Also, the fact that all inputs are present at once when they are contained by a vector makes it possible to interpret the FIR filter in a way that can be expressed with vectors:

```
fir :: Num (SExp a) ⇒ SVec a → SVec a → SVec a
fir coeff = map (dot coeff . reverse) . tail . inits
```

The new vector functions behave in the same manner as their similarly named list functions.

By wrapping the filter in a small software program that connects the filter's arguments to inputs read from standard input and result to standard output, we

can compile the filter to C. For instance, using `connectStdIO`, we can automatically connect a function like `fir` to standard input and output:

```
prog = connectStdIO $ manifestFresh . uncurry fir
```

Compiling `prog` to software yields the code in Figure 2.3 (where imports, some variable declarations and initialisation code for the second input array have been omitted for brevity).

Inspecting the generated code, we find that fusion has ensured that no intermediate arrays are created for the initial segment of `inits`. What remains are instead bits of control logic that set up the indexing bounds, which is followed by a now in-lined dot product. Its biggest flaw is perhaps that some control logic has fused its way into the inner for-loop. Co-Feldspar does however provide combinators for managing the sharing of values, for example to pre-compute the `b16` index if it gets in the way of performance. Even then, the above vector implementation only ran ∼3% slower than the original implementation in C[1] (for filters up to 10000 elements and a similar amount of coefficients).

2.4 Synchronous data-flow networks

Admittedly, a FIR filter is a fairly regular computation in the sense that each output depends on an ordered segment of previous inputs. In general, however, the output of a truly irregular recurrence relation may rely on any previous or current inputs. Such relations cannot be implemented efficiently as a sequence of array transformations, for doing so requires that we hold onto all previous inputs. For the more irregular computations, idiomatic Co-Feldspar code instead uses the signal processing library developed in this thesis.

The signal library is based around the concept of signals: possibly infinite sequences of values in some pure expression language. These signals are connected in a network that operates on streaming data, using a model of *synchronous data-flow* that is largely inspired by Lucid Synchrone (Caspi, Hamon, et al. 2008; Colaço et al. 2004), a member of the family of synchronous languages (Halbwachs 1993). The synchronous model offers high-level combinators for expressing signal-based algorithms, most of which are similar to the previous combinators for vectors. However, signals also support unit delays and their sequencing therefore carries a notion of time that is discrete and non-negative.

Internally, the signal library is a domain specific language that is embedded in Haskell and intended to extend an existing language, like Co-Feldspar, with support for synchronous data-flow. In particular, the underlying language is used to represent pure functions that, of course, can be arbitrarily complicated. The signal library then gives a means to connect such functions using a synchronous data-flow programming style.

[1] Benchmark available at: github.com/markus-git/co-feldspar/blob/master/examples/Vectors.hs

```
int main()
{
    fscanf(stdin, "%u", &v0);
    int32_t _a1[v0];
    int32_t *a1 = _a1;
    for (v2 = 0; v2 <= v0 - 1; v2++) {
        fscanf(stdin, "%d", &v3);
        a1[v2] = v3;
    }
    ...
    for (v9 = 0; v9 <= v0 - 1; v9++) {
        if (v0 <= v9 + 1) { b11 = v0; }
        else { b11 = v9 + 1; }
        if (v0 <= b11) { b10 = v0; }
        else {
            if (v0 <= v9 + 1) { b12 = v0; }
            else { b12 = v9 + 1; }
            b10 = b12;
        }
        state13 = 0;
        for (v14 = 0; v14 < b10; v14++) {
            if (v0 <= v9 + 1) { b16 = v0; }
            else { b16 = v9 + 1; }
            state13 += a1[v14] * a5[b16 - v14 - 1];
        }
        a8[v9] = state13;
    }
    fprintf(stdout, "%u", v0);
    for (v18 = 0; v18 <= v0 - 1; v18++) {
        fprintf(stdout, "%d", a8[v18]);
        fprintf(stdout, " ");
    }
    return 0;
}
```

Figure 2.3: Code listing for compilation of vector filter.

As an example of signals and their use in a synchronous paradigm, consider a function for merging a list of signals over hardware expressions by element-wise addition:

```
sums :: Num (HExp a) ⇒ [HSig a] → HSig a
sums as = L.foldr1 (+) as
```

Note that `sums` is defined in terms of the common list function `foldr1` (we prefix list functions by `L` to differentiate them from signal or vector functions with similar names), which reduces the list of signals using element-wise addition from right to left. Lists are only present in the definition of a function like `sums` to structure a signal computation and will get evaluated away before compilation.

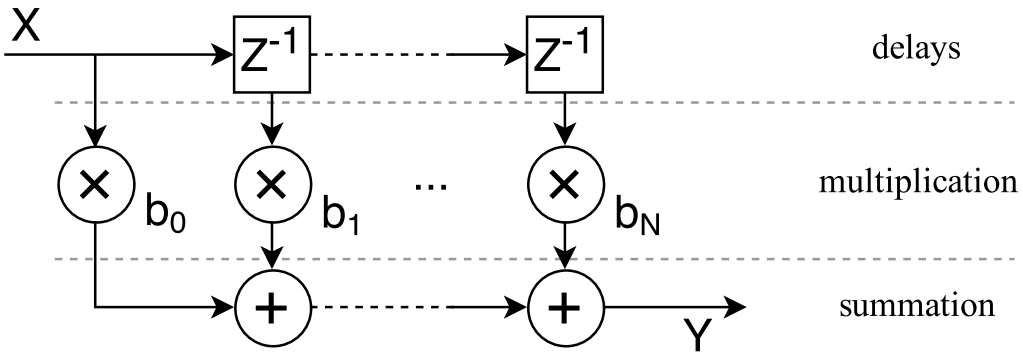

Figure 2.4: A direct form discrete-time FIR filter of order N

The summation of signals is one of the three major components making up the FIR filter. The other two are multiplication of signals with some coefficients and a number of successive unit delays of the input to tap into previous values (see Figure 2.4 for a graphical representation of a FIR filter). These two additional components can be described with standard list functions as well:

```
muls :: Num (HExp a) ⇒ [HExp a] → [HSig a] → [HSig a]
muls es as = L.zipWith (*) (L.map repeat es) as

dels :: Num (HExp a) ⇒ HExp a → HSig a → [HSig a]
dels e a = L.iterate (delay e) a
```

where `repeat` introduces a constant signal and `delay` a unit delay. These three functions together allow us to implement the full filter as follows:

```
fir :: Num (HExp a) ⇒ [HExp a] → HSig a → HSig a
fir es = sums . muls es . dels 0
```

From a hardware perspective, the above filter is arguably closer to the original specification than the vector implementation was. Furthermore, the high-level and combinatorial approach of signals provides a similarly robust programming interface.

However, in contrast to the chunking of inputs that vectors relied on, signals instead made use of potentially infinite patterns to describe the filter. For example, `dels` is implemented with `iterate`, a list function that is typically defined recursively as:

```
iterate :: (a → a) → a → [a]
iterate f x =  x : iterate f (f x)
```

That is, for an initial signal `s`, `iterate (delay 0) s` describes a list of incrementally delayed signals `[s, delay 0 s, delay 0 (delay 0 s), ...]`. While this list is potentially infinite, its composition with `muls es` ensures that we will only end up using as many delayed signals as there are elements in `es`. More specifically, it is the `zipWith` in `muls` that only requests as many signals from `dels` as there are coefficients in `es`. Lazy evaluation then ensures that only the list parts that we do end up using will be evaluated and form part of the generated network.

The above signal functions are all flat in the sense that they compile to networks without any feedback loops: `sums`, `muls` and `dels` only use recursion in their descriptions as lists, which leads to unfolding of the signal graph rather than feedback. Some interesting signal programs are however defined as a collection of mutually recursive signal functions, each built from static values or other signals. For such functions, the recursion is defined in very much the same way as recursion in common list functions. For instance, a clock pulse can be implemented as a recursive signal function, flipping its output at each instant:

```
clock :: Syntax exp Bool ⇒ Sig exp Bool
clock = true ‘delay‘ (inv clock)
```

Circular definitions like this are possible because of the `delay` applied to the recursive call.

Having defined our signal computations, we would like to peel them off and compile their synchronous data-flow graphs into efficient, imperative programs in Co-Feldspar. Signals are, however, a deeply embedded data type and their compilation is therefore not a simple de-sugaring, like it was for the shallowly embedded vectors in Section 2.3.2. To compile a signal function, we must not only translate its internal representation as signal graphs into programs, but also observe any cyclic definitions and shared signals in said graphs. This sub-problem is commonly referred to as observable sharing.

The solution to this observable sharing problem that we use is inspired by the techniques proposed by Gill (2009) and Claessen (1999), and relies on GHC-specifics to provide an `IO` function capable of directly exposing any sharing in a signal network (Aronsson and Ploeg 2013). Compilation is then fairly straightforward for shared values, as any traversal of the network can then identify the “back-edges” that it encounters.

However, for efficiency reasons, signals are not translated directly into program. Instead, a function is provided that turns signals into an imperative representation of co-iterative streams (Caspi and Pouzet 1998):

```
compile :: (HSig a → HSig b) → IO (HStr a → HStr b)
```

This version of streams breaks away from the classical one slightly and builds on a stack of two `Program` monads. The outer program is used to initialize the stream and returns the inner program, which can be executed repeatedly to produce a stream of values. As programs support stateful computations, the initial state can be updated while running; state is handled implicitly by the program monad and stream transformers are free to consume the input streams in any way they see fit. Fusion of streams is therefore achieved by construction, and even infinite streams can be handled in a strict and efficient way.

For example, consider a stream transformer that only outputs every other value:

```
odd :: HStr a → HStr a
odd (Str init) = Str $ do
  next ← init
  return $ do
    val ← next
    _   ← next
    return val
```

The new stream is initialised by running the argument's `init` function, which returns the step function `next`. A new step function is then constructed, calling `next` twice to read two inputs from the initial stream and discarding the second value. Finally, the saved input is returned as the stream's result.

As the translation of signals to imperative streams goes through the `IO` monad, signal computations should either be declared at the top level, and then threaded through a program, or in-place with the help of a "back door" into the `IO` monad. Although observable sharing in general is unsafe, the reification performed by `compile` can be used safely even with `unsafePerformIO` if some simple conditions are met. In the case of `fir`, the whole program is self-contained and it is fine to stay inside the `IO` monad and pipe its result to Co-Feldspar's interactive compiler `icompileSigH` for hardware signals:

```
design = icompileSigH =<< compile (fir [1,2])
```

Running `design` yields the description in Figure 2.5. While we limited the filter rank to keep the generated design short, we note that it follows a specific pattern where one process is generated for the combinational logic and one for managing the memory elements (Gaisler 2011).

2.5 In-place array updates

Vectors and signals provide two useful abstractions for implementing combinatorial and sequential algorithms. Both extensions define their computations as pure

```
ARCHITECTURE behav OF comp0 IS
  SIGNAL state2   : unsigned (7 DOWNTO 0) ;
  SIGNAL state2_d : unsigned (7 DOWNTO 0) :=
    "00000000" ;
BEGIN
  l8 :
    PROCESS (in0) IS
      VARIABLE v3, v4, v5 : unsigned (7 DOWNTO 0) ;
      VARIABLE v6, v7     : unsigned (7 DOWNTO 0) ;
    BEGIN
      v3 := "00000001" ;
      v4 := "00000010" ;
      v5 := resize (v3 * in0, 8) ;
      v6 := resize (v4 * state2_d, 8) ;
      v7 := resize (v5 + v6, 8) ;
      state2 <== in0 ;
      out1 <== v7 ;
    END PROCESS l8 ;
  l9 :
    PROCESS (clk) IS
    BEGIN
      IF rising_edge (clk) THEN
        state2_d <== state2 ;
      END IF ;
    END PROCESS l9 ;
END ARCHITECTURE behav ;
```

Figure 2.5: Code listing for compilation of signal filter.

functions operating on immutable values, and both provide libraries of high-level combinators that facilitate a compositional programming style. A compiler then transforms their high-level code into efficient imperative programs with mutating code. While this compositional style suits some algorithms, there are also those that rely on features that are difficult, if not impossible, to express in a combinatorial style. In particular, algorithms that rely on destructive updates of data structures can be described by neither the vector nor the signal extension.

Co-Feldspar and its functional imperative programming style supports a primitive representation of mutable arrays for which it is already possible to define in-place updates. However, the correctness of such updates is left to the designer and neither the compiler nor the type system helps prevent mistakes. The idiomatic approach for such algorithms in Co-Feldspar is therefore to employ the new array programming language developed in this thesis. The new arrays offer safe, purely functional and crash-free in-place array transformations. The array library not only supports a functional imperative programming style, but also provides a collection of high-level combinators and abstractions for pure and efficient array computations that fully supports equational reasoning.

Code written using the new array library differs slightly from the previous vector- and signal-based approaches, for the programmer is now in control over features that were previously left to the compiler. In particular, the array library provides a primitive `vcopy` instruction for creating *virtual copies* of arrays. Virtual copying is the mechanism that allows users to express safe re-use of arrays, as it gives the illusion of having copied a value. However, no such copy is ever made: `vcopy` only makes a safe alias of the array.

The combinator library that comes with the new arrays provides a number of convenient functions for creating virtual arrays, most of which are based on functions for creating mutable arrays in Co-Feldspar. This means that modifying computations to run in-place is straightforward: The programmer first writes their algorithm in a pure and functional style, they then specify what computations should be done in-place by swapping a fresh array or copy for a virtual one.

As an example of programming with virtual copies, consider a short software program that multiplies two vectors:

```
mul :: SVec Int32 → SVec Int32 → S (SVec Int32)
mul arr brr = manifestFresh (zipWith (*) arr brr)
```

The `manifestFresh` combinator evaluates its vector argument and stores the result into a freshly-allocated array. In order to make this function run in-place, for instance by writing its result back into `arr`, we only need to replace `manifestFresh` for a combinator that instead produces a virtual copy of `arr`. We modify `mul` as:

```
mul :: SVec32 → SVec32 → S (SVec32)
mul arr brr = manifestReuse arr (zipWith (*) arr brr)
```

The `manifestReuse` combinator evaluates its second argument and stores the result

into the array given in the first argument, effectively re-using the `arr` array instead of allocating fresh storage. The compiler checks that this storage re-use is safe, that is, it checks that the program behaves as if a fresh array had been allocated.

Internally, both `manifestFresh` and `manifestReuse` are implemented with the same `manifest` combinator for storing vectors into arrays:

```
manifest :: Comp m ⇒ Arr m a → Vec (Exp m) a
  → m (Arr m a)
manifest arr (Pull len f) =
  for (0, 1, min (length arr) len)
      (λi → setArr arr i (f i))

manifestFresh pull = do
  arr ← newArray (length pull)
  manifest arr pull

manifestReuse arr pull = do
  assert (length arr == length pull)
  copiedArr ← vcopy arr
  manifest copiedArr pull
```

There is however one crucial difference between the two functions: `manifestReuse` creates a virtual copy of `arr` with its `vcopy` statement. Given an array `a`, `vcopy a` creates an array `b` that gives the illusion of being a copy of `a`. Semantically, it behaves as if we had created a fresh array `b` and copied the values of `a` into `b`; users can mentally replace all occurrences of `manifestReuse` with `manifestFresh` when reasoning about the program.

For regular computations that consume their input in order, such as `mul`, the virtual arrays can be reused without much worry. Computations that are not as regular require some care because they run the risk of overwriting values that will be read later on, which would break the illusion of a virtual copy. For instance, it is not possible to reverse a vector in-place if it is traversed in the usual way. A common solution is instead to traverse the array up to its half-way point, updating elements at both the current index and the mirrored index at the opposite side simultaneously. This pattern is actually quite common, and Co-Feldspar therefore provides a `pairwise` combinator that implements it:

```
pairwise :: Comp m ⇒ Exp m Index
  → (Exp m Index, Exp m Index)
  → Vec (Exp m) a → Push m a
pairwise ixs vec = Push (length vec) $ λwrite →
  for 1 (length vec) $ λi → do
    let (ix1, ix2) = ixs (i-1)
    iff (ix1 ≥ ix2) (return) $ do
      x ← shareM (vec ! ix1)
      y ← shareM (vec ! ix2)
      write ix1 x
      write ix2 y
```

where `Push` is a vector type that encodes its own iteration scheme (Claessen, Sheeran, et al. 2012), and can conceptually be thought of as a program that writes an array to memory using a given `write`-function.

We can use `pairwise` implement a reverse in software as follows:

```
reverse :: SArr Int → S (SArr Int)
reverse arr = manifestReuse arr (rev arr)
  where len = length arr
        rev = pairwise (λix → (ix, len-ix-1))
```

Please note that Co-Feldspar does not attempt to check the safety of programs at the target or source level. Instead, it first elaborates away any high-level instructions and introduces its safety assertions; it then verifies the safety of this intermediate code. The principal idea is that correctness is ensured by using an off-the-shelf-theorem prover to discharge the safety conditions introduced during this elaboration. This step is performed before compilation, so if the prover is able to show correctness then no assertions end up in the generated code.

2.6 Memory management with regions

Co-Feldspar adopted a monadic interface to combine explicit memory management with its otherwise pure semantics; pure expressions alone cannot efficiently express algorithms that, for performance, rely on a particular evaluation order or in-place updates. This separation into pure and impure parts is, however, a bit crude and imposes a fairly low-level style of programming to describe only a select class of algorithms. While this problem is alleviated by the aforementioned vector, signal and array libraries, imperative programs and explicit memory management still make up the core of Co-Feldspar.

Even with explicit allocation, it can be difficult for users to know the exact lifetime of their values. Consider as an example the following code snippet, which introduces and returns a reference from within a local statement:

```
escape :: Software (SRef Int32)
escape = when (true) (do r ← newRef; return r)
```

To compile `escape` in a stack-based memory management discipline (like that of Co-Feldspar) we must move the allocation of `r` to the outer scope. This movement can lead to references ending up in larger blocks in the source code than expected, and if the outer block is especially large, that memory may outlive its usefulness by some margin

It is because of the aforementioned issues that we explore the use of a memory management discipline based on regions for embedded software.

The beauty of the region discipline is that it both relieves users of explicit memory management and has the potential for excellent runtime space usage, even without other forms of garbage collection. A region discipline uses a memory model where the store can be thought of as a stack of regions, where regions themselves are like stacks of unbounded size. Each region grows in size as more values are allocated in it, until it is popped off the region stack in its entirety. These regions are allocated and de-allocated at inferred points, and every value-producing expression is labelled with the region its result should be stored in.

Unfortunately, however, labelling values with as small as possible regions is a rather complicated process. Much of the complexity comes from how difficult it is to perform an accurate lifetime analysis for values in higher-order functions, particularly in those that call themselves recursively. But a region labelling is only as effective as its lifetime analysis allows, so languages that implement regions typically rely on elaborate type systems to track values in expressions. Indeed, region inference as first presented by Tofte and Talpin (1994) used ideas from effect inference to build a type-and-effect system for inferring regions.

The accuracy that type-and-effect systems offer comes at a cost of additional kinds for regions and effects at the type level, as types are labelled with the regions they touch. For example, the successor function $succ\, x = x + 1$ can be assigned the following type scheme after region inference:

$$\forall \rho_1, \rho_2.\ (\mathrm{Int}, \rho_1) \xrightarrow{\{\mathbf{get}\,\rho_1, \mathbf{put}\,\rho_2\}} (\mathrm{Int}, \rho_2)$$

The effect that *succ* is "getting" the argument from region ρ_1 and "putting" the result into region ρ_2 is recorded by the **get** and **put** constraints, respectively. In general, a computation that accesses a region ρ adds $\mathbf{get}\,\rho$ to its associated effect set; conversely, any computation that stores values into region ρ adds $\mathbf{put}\,\rho$. Even with the help of effects, however, region inference struggles with recursion in the special cases of tail recursion and iteration (Tofte, Birkedal, Elsman, and Hallenberg 2004).

There are however domains that reject the use of recursion because it is difficult to predict the time and space costs of its evaluation. For example, predictable performance is a crucial feature for embedded software, since it typically runs on resource constrained devices. We therefore believe that a region inference that specifically targets embedded software could avoid most of the complexities in traditional type-and-effect systems and still permit us to express interesting programs.

We explore a region inference that is limited to first-order values and inspired by qualified types (Jones 2003) rather than type-and-effect systems.

The essential idea behind a system of qualified types is to include a language of predicates that restricts the instantiation of types. For example, if we define a predicate $\mathbf{Put}\,\rho$ on regions that is true if ρ is allocated, then we can construct a type scheme $\forall\rho, \tau.\, \mathbf{Put}\,\rho \Rightarrow \tau$ to represent the set of types:

$$\{\tau \mid \tau \text{ is a type and } \rho \text{ is an allocated region}\}$$

An object that has been assigned the above type can then only be used as an object of type τ if it is also supplied with suitable evidence that ρ is indeed allocated. Such witnesses are provided by a language of evidence expressions with their own kind of abstraction and application. For instance, a term $E : \mathbf{Put}\,\rho \Rightarrow \tau$ can be generalised as $\lambda w.E$, where $w : \mathbf{Put}\,\rho$ is an evidence variable.

Note that a qualified type $\mathbf{Put}\,\rho \Rightarrow \tau$ in a sense captures the effect of an expression "putting" values into region ρ. It then follows naturally that evidence for such predicates would be a fresh location in memory, which the labelled object can then store its value in. A complete region labelling is then formed by simply marking each value-producing object with such a qualified type. More specifically, we formulate our region labelling as a translation from a well-typed source language into a mostly similar target language, except for certain region annotations. Two annotations in particular are introduced for terms:

$$E\,\mathbf{at}\,w \quad \text{and} \quad \mathbf{letregion}\,w\,\mathbf{in}\,F$$

The first is used whenever E directly produces a value, and indicates that its value should be stored in the region bound to w. The second creates a region corresponding to w before F is entered, and values can be allocated within that region for the duration of the expression; the region is reclaimed once F is evaluated.

Annotations are also introduced at the type level, where value types are tagged with a region ρ and named predicate $w : \mathbf{Put}\,\rho$, binding ρ to an object's place tag w. For example, a pair of integers $(1, 1)$ is translated into a labelled pair $(1, 1)\,\mathbf{at}\,w : (\mathrm{Int}, \mathrm{Int})^{\rho}$ and a predicate $w : \mathbf{Put}\,\rho$. These predicates are collected in a context during inference, and record the effects of an expression as a whole rather than its individual components.

While convenient for type checking, collecting region predicates in a context makes it difficult to check what regions a particular term manipulates. So, how can we tell whether the introduction of a local region is premature or not? Surely, if an expression is given a primitive type like Int in a strict language, then all memory allocated during the computation of the integer could be de-allocated once the result has been computed (except for the memory that holds the integer) and thus stored in local regions.

A first-order type τ similarly represents a fully computed value in the sense that it is assigned to objects that do not include any closures. If such an object E has produced a predicate $\mathbf{Put}\,\rho$, where ρ cannot be reached from either the type τ or from the context of variable assignments, then we could hypothetically generalise

it as $\lambda w.E : \forall\rho.\,\mathbf{Put}\,\rho \Rightarrow \tau$. Substitutions could certainly introduce higher-order types (and inject closures into E), but it is obvious that any such types have no opportunity to refer to the universally quantified region ρ (just as no closures in the context can refer to its associated witness w). As a result, any memory associated with w could safely be de-allocated once the result of E has been computed.

We should note that the above hypothetical type scheme of $\forall\rho.\,\mathbf{Put}\,\rho \Rightarrow \tau$ is ambiguous according to Jones' terminology for qualified types, since ρ does not appear in τ and cannot therefore be determined by the context. Systems that implement qualified types generally attempt to apply defaulting for such cases, in order to produce a deterministic witness for the predicate if possible. But in the proposed region system, there is only one way of creating evidence variables (by allocating new memory), so defaulting is always possible. In fact, the inference of a local region can be done by creating a suitable witness w for a predicate $\mathbf{Put}\,\rho$ such that the above type scheme can immediately be instantiated. The result then becomes an object of type τ, but where the witness w is given by a **letregion** and no longer bound.

As an example, consider a pair of integer values that we have labelled with regions as $(\alpha, \beta)\,\mathbf{at}\,w : (\mathrm{Int}, \mathrm{Int})^{\rho}$ and given a predicate $w : \mathbf{Put}\,\rho$. If we take the 2^{nd} projection of such a pair, the result is simply an integer $\beta : Int$ with no mention of its internal region ρ. In fact, no reference to ρ can escape β since Int is also a first-order type. We could therefore resolve the predicate $w : \mathbf{Put}\,\rho$ and introduce a local region to the expression as $\mathbf{letregion}\,w\,\mathbf{in}\,\mathbf{snd}(\alpha, \beta) : \mathrm{Int}$. Future substitutions may change both α and β as well as their types, but regardless of how these changes are done they cannot depend on the local region.

According to the above reasoning, the first-order restriction is really too strong; it is enough that a term's principal type is first-order for the system to draw conclusions about what region links it carries. In fact, even this restriction is too strong. It is the principal type in an abstract context (where every parameter is mapped to a type variable) that is the most interesting one.

However, the cost of limiting regions to first-order terms is not as severe a restriction on the kind of programs that we can efficiently label as it might first appear, especially since the restriction primarily affects those programs that deliberately keep memory-heavy objects in scope. As an example, consider the following code:

```
f = (let p = (λx. x, Heavy) in fst p)
```

Here, the memory heavy computation *Heavy*, is deliberately hidden behind a higher-order function and paired up with another value so that it cannot be discarded. As a result, any program that mentions f would be forced to keep *Heavy* around in memory until f is completely evaluated, which could potentially take some time.

Whether this new region inference struggles with labelling certain kinds of programs remains to be seen, but we nevertheless feel that region inference based on qualified types provides a succinct account of region-based memory management for embedded systems. The restriction to languages without recursion also provides some benefits that cannot be implemented in the original system. For example, we

are able determine the size of regions statically. An implementation of the current inference system is available as open source[2].

2.7 Related work

Sheeran pioneered the use of functional languages in hardware design with μFP (Sheeran 1984), a language that uses functional combinators to describe complex hardware from a set of smaller circuits and gates. The Lava family (Bjesse et al. 1998; Gill et al. 2010; Naylor 2009) of functional languages have since expanded upon the ideas of *mu*FP and introduced modern functional features. For instance, Lava exploits monads and type classes to provide multiple interpretations of circuit descriptions, such as simulation, formal verification and generation of netlists. Polymorphism and higher-order functions provide general descriptions of hardware designs. In a sense, Co-Feldspar and the signal library have further extended this family to heterogeneous systems, where control logic is typically implemented in software rather than hardware.

Outside of the Lava family, there is Bluespec (Nikhil 2004), a hardware description language that is influenced by functional languages and includes, for instance, higher-order functions and polymorphism. In contrast to Lava, Bluespec can describe both software and hardware. Nevertheless, Bluespec descriptions are written at a clock-cycle granularity and therefore provide a lower level of abstraction than most functional languages. A third example is Chisel (Bachrach et al. 2012b), a hardware description language embedded in Scala, which, like Bluespec, supports both cycle-accurate software simulation and hardware generation from its descriptions.

Compiling an ordinary C program to a hardware description has great appeal, but finding a translation between the two has however proven to be difficult. Tools like Catapult C (Mentor Graphics 2008) are able to generate register transfer level code from ordinary C descriptions, but its sequential programs are often a bad fit for the parallelism inherent to most hardware architectures. Additional C based attempts includes the creation of SystemC (Ghenassia et al. 2005), a set of classes and macros that provide an even-driven simulation interface in C++. Although strictly a C++ library, SystemC is often viewed as a language of its own that simply reuses C++ syntax. Semantically, it has quite a few similarities to VHDL and Verilog.

Cryptol (Browning and Weaver 2010) is a DSL for the specification of cryptographic algorithms, and can generate both C and VHDL/Verilog from the same description. While not an embedded language, Cryptol has similar ambitions to the co-design language, in particular the ability to do rapid development and design exploration—although the latest versions of Cryptol no longer support hardware generation. Cryptol is however a standalone language, so any extension to it cannot benefit from the ecosystem of tools available in the host language. Another language outside the domain of HLS is Microsoft's Accelerator (Tarditi et al. 2005), for programming GPUs and various other platforms. Accelerators provides a high-level data-parallel programming model as a library that is available from a conventional

[2] Available at: github.com/markus-git/regions

imperative programming language like C. However, the project seems, unfortunately, to be no longer active.

There exist several methods that aid in the embedding of languages in Haskell. Among these different approaches is finally tagless (Carette et al. 2009), which associates each group of language constructs with a type class, and each interpretation with a semantic domain type. The result is an expressive and compositional way of embedding languages. This does, however, come at a cost of awkward types—the type of an embedded language is simply a qualified type variable—and it tends to expose implementation details to users. The abstract types of Co-Feldspar hides such details, but are fixed at the level of arrays, variables and signals. Another interesting embedding technique to consider for adoption is thus high-level views on low-level representations (Diatchki et al. 2005), which provides bit-level views of data types in functional languages.

For Scala, the Delite (Sujeeth et al. 2014) library provides a framework of reusable components for embedded languages, like optimizations, and code generators. Delite produces an intermediate representation of user programs that is similar to the model used by the co-design language, but targets a combination of CPU and GPU systems. Lightweight Modular Staging (LMS) has also been explored as an option to ease the construction of a domain-specific High Level Synthesis (HLS) system (George et al. 2013a) in Scala. The argument is that LMS eases the reuse of modules between different HLS flows, and makes it easier to link to existing tools, such as the C compilers that are able to produce register transfer level descriptions. Though the language-specific challenges for LMS are different from those faced by the co-design language, the two approaches are comparable in terms of capability. The way that code generation of programs is built upon the translation of monads to imperative programs is also reminiscent of Sunroof (Bracker and Gill 2014), a DSL for generating JavaScript, and MIL (Jones et al. 2018), a monadic intermediate language for implementing functional languages.

The signal library presented in this thesis is based on the synchronous data-flow paradigm and is inspired by other languages from this domain. Of the synchronous languages, Lucid Synchrone (Caspi, Hamon, et al. 2008; Colaço et al. 2004) is perhaps our biggest source of inspiration and provides a synchronous functional programming for reactive systems. Initially, the language was introduced as an extension of LUSTRE (Halbwachs 1993), and extended the language with new and powerful features. For instance, automatic clock and type inference were introduced and a limited form of higher-order functions was added. Lucid Synchrone is however a standalone language, and thus cannot be easily integrated with the co-design language. Zélus (Bourke and Pouzet 2013), a successor of Lucid Synchrone, has shown that synchronous languages can be extended to model hybrid systems as well, that is, systems that consist of both continuous and discrete components.

The array library is similar, in a sense, to Futhark (Henriksen et al. 2017), a purely functional data-parallel array language that excels at GPU programming. Futhark employs a simple type system with uniqueness labels that, together with a restricted language of primitives, facilitates in-place updates of arrays. The futhark compiler verifies that a copied array and its aliases are not used on any execution

path following the in-place update. Virtual array copies, in contrast, only required that the original array is not used in a way that could break the illusion of the copy.

Ivory (T. Elliott et al. 2015) is functional language that is embedded in Haskell and has support for model-checking, theorem proving and property-based testing. Ivory uses these features to enforce memory safety for its programs, and tries to avoid most undefined behaviour while still providing low-level memory control. In particular, a sort of type-level region labels are used to ensure that memory references do not persist beyond the scope of their containing monadic computation.

Another "safe-C" language with a similar goal is Copilot (Pike et al. 2010), a hard real-time and data-flow language that generates small and efficient C programs with embedded runtime monitors and their own scheduler. However, Copilot focuses on expressing pure computations, and does not provide as convenient support for defining new data-structures or manipulating memory as Ivory or Co-Feldspar. Timber (Black et al. 2002) is an imperative object-oriented and purely functional language that is also designed for real-time embedded systems, but offers message passing primitives for both synchronous and asynchronous communication between concurrent and reactive objects.

Another common approach to handling in-place updates is to employ various type systems. One such system is linear types (Wadler 1990), where values belonging to a linear type must be used exactly once; linear types cannot be duplicated or discarded. Such values can safely admit in-place updates. Another approach is adopted by Rust (Matsakis and Klock II 2014), a safe C language for developing reliable and efficient programs. Rust's static and affine type system is safe and expressive and prevents pointer aliasing errors as well as providing strong guarantees about isolation, concurrency, and memory safety. Rust has a framework for property based testing, but lacks support for verification and static analysis of its programs. Spark/Ada (Barnes 2003) is a language similar to Rust and provides high-assurance embedded programming, with a contract language and verification tools to prove invariants. While the language is programmed at a higher-level than C, it is also more restrictive than Rust; in particular, there are no references in Spark.

Safe languages are not exclusively functional, and the Cyclone project (Jim et al. 2002) has done pioneering work to create a safe dialect of C. Cyclone is designed from the ground up to prevent the buffer overflows and memory management errors that are common in C programs, while retaining C's syntax and semantics. Most of Cyclone's language design indeed comes directly from C; Cyclone uses the C pre-processor, and, with few exceptions, follows C's lexical conventions and grammar. However, Cyclone does not provide macro-programming facilities beyond its C pre-processor.

Another interesting feature of Cyclone is its use of region-based memory management (Grossman et al. 2002). Specifically, Cyclone provides three kinds of regions: a single heap region, stack regions, which correspond to local definitions in C, and dynamic regions, which binds the lifetime of a region to the execution of a statement. The region discipline provided by Cyclone is a variation of the region typing discipline introduced by Tofte and Talpin (1997) for C, and relies on a novel type-and-effect system for tracking region links. Cyclone requires that users write some region annotations, but attempts to reduce this burden by providing default annotations

and a simplified treatment of region effects.

ML Kit (Tofte, Birkedal, Elsman, Hallenberg, et al. 2001), a version of Standard ML, is a functional language that also uses automatic region inference in place of traditional garbage collection. ML Kit has operations for allocating, de-allocating and extending regions, but also provides an explicit operation for resetting an existing region by reclaiming its memory without removing it from the region stack. The latter annotation was introduced in part to help users address the inefficient memory usage that occurred in some recursive programs (Tofte, Birkedal, Elsman, and Hallenberg 2004). ML Kit has also been extended with a garbage collector to further improve its region pruning.

2.8 Discussion

Heterogeneous computing represents an interesting development in the domain of embedded systems, as these systems incorporate more than one kind of processor to handle particular tasks. The performance of these systems comes at a cost of increased programming complexity, as the level of heterogeneity can introduce non-uniformity in both development and overall system capabilities. Heterogeneity in such a system, combined with the use of low-level languages, leads to designs with low re-usability and modularity; it is difficult to provide useful abstractions when types are limited to scalar values, composite data type structures and pointers or signals.

Many of the aforementioned issues with low-level languages are related to lack of modularity. Some are directly related, like reusable functionality and architecture, others only indirectly. We assert that many of these problems can be addressed by features from functional programming and therefore developed the Co-Feldspar language, a functional and domain-specific language for hardware software co-design.

The Co-Feldspar language presented in this thesis represents our efforts to create a functional language that allows modular and generic implementations of heterogeneous programs for embedded systems, while still providing a syntax and semantics that are designed to give predictable performance. In particular, Co-Feldspar makes extensive use of higher-order functions to build reusable combinator libraries and type classes to support programs with overloaded instructions that can be interpreted as both software and hardware programs. Exploiting these functional features makes it possible to define Co-Feldspar, a staged language with a compiler that is capable of generating both C and VHDL functions, which can deployed on different processing elements that communicate via AXI interconnects.

At its core, Co-Feldspar is an imperative language with a monadic interface that provides simple instructions, expressions and explicit memory management. However, the use of a mixed shallow and deep embedding approach makes it straightforward to add new features and interpretations. For example, both the signal and array libraries were added to Co-Feldspar without changing its internal representation of programs or its compiler. Both libraries provide a comfortable syntax for problems in their respective domains, and support an efficient translation into the relatively simpler programs in Co-Feldspar. The signal library allows users to build their

programs by composing high-level combinators, and its compiler then transforms this high-level code into efficient, low-level, imperative streams. Similarly, the array library provides high-level combinators that enable a functional approach to defining in-place array algorithms.

An interesting extension for Co-Feldspar lies in the definition of a systems description layer. This layer would provides a means of tailoring programs to a particular embedded system at compile time by, for instance, distributing and orchestrating a function on a heterogeneous multi-core accelerator.

While powerful, Co-Feldspar's monadic interface for mutable references and arrays is a rather low-level interface with explicit indexing. This thesis therefore explores a more automatic memory management discipline with regions, which we have tailored specifically to embedded systems. The region labelling is intended to augment the current stack-based memory allocation of Co-Feldspar so that users only need to worry about memory management in algorithms that rely on heap allocation or in-place updates. However, the beauty of our new region labelling lies in its simplicity, as regions are inferred by a system of qualified types rather than a type-and-effects system.

Work is currently ongoing to define a practical implementation of a compiler for programs labelled by our new region system. Having such an implementation will clearly make a strong argument for the usefulness of a restricted approach for region labelling, and provide what we believe to be a useful alternative to the monadic interface of Co-Feldspar.

2.9 Summary of contributions

Recall that we originally set out to explore the benefits of functional programming in the development of heterogeneous embedded systems, in particular the necessity of an extensible core, flexible interpretation, and intuitive and efficient abstractions. A natural question, then, is how successful the presented techniques are in addressing these points? We repeat the research questions here:

1. Can the embedded, heterogeneous system development for modern FPGAs benefit from a functional hardware software co-design language? In particular:

   (a) Can a low-level, core language be embedded within a functional host to provide a model of the various imperative languages used to describe heterogeneous systems that is both extensible and has a relatively small semantic gap to C and VHDL?

   (b) Can such an embedded core language exploit the higher-order functions and rich type systems of its host to separate the syntax of its programs from their semantics such that a program can be parameterised by the functionality it requires and, once written, can be interpreted in a variety of ways?

2. How can imperative functional programming be extended with support for efficient, synchronous data-flow definitions, reducing the gap between streaming algorithms and their implementation in an otherwise imperative language?

3. Can a functional array programming language offer safe, in-place array transformations without neither weakening its transparency nor the ability to apply equational reasoning?

4. Does a functional language without recursion permit the definition of a region-based memory management scheme that is simpler than standard type-and-effect systems for region inference?

Hardware software co-design In order to address Research Question 1, Co-Feldspar was designed to be modular in the perspectives of both an implementer (1a) and a user (1b).

From the language implementer's perspective, the mixture of shallow and deep embeddings in Co-Feldspar's core makes it easy to extend its programs with new primitives and interpretations. For example, Papers 1 and 3 managed to add support for synchronous streams and virtual array copies without having to change the Co-Feldspar language or compiler. The performance of such abstractions and other, sophisticated instructions was ensured by having them be accompanied by a translation into programs with only simpler instructions. This compilation scheme not only safeguards against common errors (Axelsson 2016), but also provides language implementers with a great deal of control over the code generation processes; compilation and elaboration are explicit operations, independent of running the compiled code.

From the users' perspective, it is possible to write modular and generic code through Co-Feldspar's costless abstractions and hierarchy of type classes, which defines overloaded types and instructions that can be used in both hardware and software programs. Practically, these type classes provides an extensible and structured way of controlling the interpretation of user defined programs; type classes enable us to reason about the presence, and absence, of certain instructions in a program. For example, in Section 2.3.2, the FIR filter was constructed using generic expressions and combinators from the vector library. Since the vector library guarantees fusion of its operations, the use of combinators incurred neither any intermediate arrays, nor any extra traversals.

The above points mostly touch on the construction of generic and hardware or software specific programs. In the case that programs cross the software hardware boundary, the user must first establish an interface between the offloaded program and its external environment. These interfaces are currently given by a small language of programmable signatures that lets the programmer describe the mapping of each argument to a program. While this approach gives the programmer a great deal of control over the communication, such an interface must be defined manually for each communication channel. Once defined, however, a signature can be automatically fitted to an AXI4-lite interconnect and synthesised.

Practically, we have tested the usefulness of Co-feldspar by implementing FIR, IIR and FFT filters, as well as a larger cryptographic example (Moriarty 2017). In general, we found that Co-Feldspar's monadic interface and vector library do provide predictable performance; fusion of vectors ensures that functions are aggressively in-lined and that generated loop structure is satisfactory with a shape that is easy to influence. Further, explicit memory management avoids unnecessary copying.

Signal processing In regards to Research Question 2, we have found that the signal library provides a comfortable syntax for synchronous data-flow. Furthermore, the representation of signals as imperative, co-iterative streams yields an intuitive semantics that can be efficiently represented as programs in Co-Feldspar. In particular, the mixture of deeply embedded signals and reification allowed the library to provide combinators that retain much of the original elegance and modularity of a purely functional programming style and still support efficient compilation.

During testing we found that a signal implementation of the FIR and IIR filters preformed slightly better than their reference implementations in C, at least up to filter ranks of fifty. The drop off in performance at that point can be explained in part by the construction of the filter: using lists to describe a filter, like in Section 2.4, results in unrolling of the entire signal graph, which in turn can result in loop structures that are too big to run efficiently. A different trade-off between time and space could have resulted in a more manageable code size at the cost of some latency. Such trade-offs are already supported by the signal library, as we can replace the recursion in Haskell lists by recursively defined signals.

Safe in-place updates The array programming library explored Research Question 3 and offers safe, purely functional and crash-free in-place array transformations. It manages to do so by adding one crucial feature: virtual copying.

Virtual copying is the mechanism that allows the array programming library to provide a high-level, functional approach to defining an array algorithm that can execute selected parts of its computation safely in-place; users define computations as normal and then substitute certain arrays for virtual copies. Since the compiler checks that performing these in-place updates does not change the behaviour of the computation, it is still pure. Users can therefore freely use equational reasoning to understand and develop their programs.

In order to evaluate the usefulness of virtual array coping, we compared the implementation of an in-place FFT with virtual array copies against a previous implementation that relied on unsafe functions to express its in-place updates (not checked by a compiler). The combinators from our new array programming library allowed us to re-implement the in-place FFT and verify its safety, all without having to disable any safety checks in the compiler. In fact, the verifier was able to discharge all but the assertion that the input had length a power of two. The combination of in-place updates and bounds check removal increased performance by 30% in some cases. A similar performance could only be achieved in the previous implementation by compiler directives to disable the generation of internal safety and bounds checks.

Memory management with regions In the final part of this thesis we explored a region inference for embedded systems that attempt to solve Research Question 4. In particular, we developed a simplified, region-based memory management scheme by taking inspiration from qualified type systems.

Because our new region inference is restricted to languages without recursion, much of the standard complexity in inferring regions with type-and-effect systems (such as polymorphic recursion in regions) could be avoided altogether. And by approximating the condition on which a region is referenced by higher-order functions, we can further simplify the inference rules by avoiding most of the labelling of functions with their latent effects. These restrictions are not as severe as one might think, and even introduce some new benefits. For instance, we can always derive a total memory cost of an expression from its annotations. This allows users to make informed decisions about how they need to structure programs for memory efficiency, and if at all necessary to do so. While the restructuring of programs to be more region friendly usually requires some familiarity with the inference systems, we would argue that doing so should be considerably easier with a system based on qualified types than with a type-and-effect system.

Bibliography

Apfelmus, H. (2016). *The operational package, version 0.2.3.2*. URL: `https://hackage.haskell.org/package/operational` (cit. on p. 13).

Aronsson, M. (2014). "Streaming Computations in Feldspar". MSc Thesis. Dept. of Computer Science & Engineering, Chalmers University of Technology (cit. on p. 10).

Aronsson, M., E. Axelsson, and A. Persson (2019). *Imperative-EDSL*. URL: `https://hackage.haskell.org/package/imperative-edsl` (cit. on p. 13).

Aronsson, M. and A. van der Ploeg (Oct. 2013). *Simple observable sharing*. URL: `http://hackage.haskell.org/package/observable-sharing` (cit. on p. 31).

Axelsson, E. and A. Persson (2015). "Programmable Signatures". In: *Trends in Functional Programming, Revised Selected Papers*. Springer (cit. on p. 22).

Axelsson, E. (2016). *Compilation as a Typed EDSL-to-EDSL Transformation (Blog post)*. `http://fun-discoveries.blogspot.se/2016/03/` (cit. on pp. 19, 45).

Axelsson, E. (2019). *A version of Operational suitable for extensible EDSLs*. `http://hackage.haskell.org/package/operational-alacarte` (cit. on p. 13).

Axelsson, E. et al. (Aug. 30, 2016). *Resource-Aware Feldspar*. URL: `hackage.haskell.org/package/raw-feldspar` (cit. on p. 7).

Axelsson, E., K. Claessen, G. Dèvai, Z. Horvàth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda (July 2010). "Feldspar: A Domain Specific Language for Digital Signal Processing algorithms". In: *Formal Methods and Models for Codesign*. IEEE Computer Society, pp. 169–178 (cit. on p. 6).

Axelsson, E., K. Claessen, M. Sheeran, J. Svenningsson, D. Engdal, and A. Persson (2011). "The Design and Implementation of Feldspar". In: *Implementation and Application of Functional Languages*. Ed. by J. Hage and M. Morazán. Vol. 6647. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 121–136. ISBN: 978-3-642-24275-5 (cit. on pp. 6, 17).

Axelsson, E., A. Persson, and J. Svenningsson (2014). *Feldspar's Streams*. URL: `https://github.com/Feldspar/feldspar-language/blob/master/src/Feldspar/Stream.hs` (cit. on p. 10).

Baaij, C., M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards (2010). "CλaSH: structural descriptions of synchronous hardware using Haskell". In: *Proc. 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD)*. IEEE (cit. on p. 16).

Bachrach, J. et al. (2012a). "Chisel: Constructing Hardware in a Scala Embedded Language". In: *Proc. 49th Design Automation Conference (DAC)*. ACM (cit. on p. 16).

Bachrach, J. et al. (2012b). "Chisel: Constructing Hardware in a Scala Embedded Language". In: *Proceedings of the 49th Annual Design Automation Conference*. ACM, pp. 1216–1225 (cit. on p. 40).

Barnes, J. G. P. (2003). *High integrity software: the Spark approach to safety and security*. Pearson Education (cit. on p. 42).

Bjesse, P., K. Claessen, M. Sheeran, and S. Singh (1998). "Lava: Hardware Design in Haskell". In: *ACM SIGPLAN Notices*. Vol. 34. 1. ACM, pp. 174–184 (cit. on pp. 16, 40).

Black, A. P., M. Carlsson, M. P. Jones, R. Kieburtz, and J. Nordlander (2002). *Timber: A programming language for real-time embedded systems*. Tech. rep. Technical Report CSE-02-002, Dept. of Computer Science & Engineering, Oregon ... (cit. on p. 42).

Bourke, T. and M. Pouzet (Mar. 2013). "Zélus: A Synchronous Language with ODEs". In: *16th International Conference on Hybrid Systems: Computation and Control*. ACM, pp. 113–118 (cit. on p. 41).

Bracker, J. and A. Gill (2014). "Sunroof: A Monadic DSL for Generating JavaScript". In: *Proc. 16th Int. Symp. on Practical Aspects of Declarative Languages*. Springer International Publishing, pp. 65–80. ISBN: 978-3-319-04132-2 (cit. on p. 41).

Browning, S. and P. Weaver (2010). "Designing tunable, verifiable cryptographic hardware using Cryptol". In: *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer, pp. 89–143 (cit. on p. 40).

Carette, J., O. Kiselyov, and C.-c. Shan (2009). "Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages". In: *Journal of Functional Programming* 19.5, pp. 509–543 (cit. on p. 41).

Caspi, P., G. Hamon, and M. Pouzet (2008). "Synchronous functional programming: The lucid synchrone experiment". In: *Real-Time Systems: Description and Verification Techniques: Theory and Tools. Hermes* (cit. on pp. 28, 41).

Caspi, P. and M. Pouzet (1998). "A Co-iterative Characterization of Synchronous Stream Functions". In: *Electronic Notes in Theoretical Computer Science* 11.0, pp. 1–21. ISSN: 1571-0661 (cit. on p. 31).

Chung, E. S., P. A. Milder, J. C. Hoe, and K. Mai (2010). "Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs?" In: *2010 43rd annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, pp. 225–236 (cit. on p. 15).

Claessen, K. and D. Sands (1999). "Observable Sharing for Functional Circuit Description". English. In: *Advances in Computing Science — ASIAN'99*. Ed. by P. Thiagarajan and R. Yap. Vol. 1742. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 62–73. ISBN: 978-3-540-66856-5 (cit. on p. 31).

Claessen, K., M. Sheeran, and B. J. Svensson (2012). "Expressive Array Constructs in an Embedded GPU Kernel Programming Language". In: *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*. ACM, pp. 21–30 (cit. on p. 36).

Colaço, J.-L., A. Girault, G. Hamon, and M. Pouzet (2004). "Towards a Higher-order Synchronous Data-flow Language". In: *Proceedings of the 4th ACM International*

Conference on Embedded Software. EMSOFT '04. Pisa, Italy: ACM, pp. 230–239. ISBN: 1-58113-860-1 (cit. on pp. 28, 41).

Diatchki, I. S., M. P. Jones, and R. Leslie (2005). "High-level views on low-level representations". In: *ACM SIGPLAN Notices* 40.9, pp. 168–179 (cit. on p. 41).

Elliott, C., S. Finne, and O. De Moor (May 2003). "Compiling Embedded Languages". In: *Journal of Functional Programming* 13.3, pp. 455–481. ISSN: 0956-7968 (cit. on p. 16).

Elliott, T., L. Pike, S. Winwood, P. Hickey, J. Bielman, J. Sharp, E. Seidel, and J. Launchbury (2015). "Guilt free ivory". In: *ACM SIGPLAN Notices*. Vol. 50. 12. ACM, pp. 189–200 (cit. on p. 42).

Feist, T. (2012). *White Paper: Vivado Design Suite*. Tech. rep. Xilinx. URL: `https://www.xilinx.com/products/design-tools/vivado.html` (cit. on p. 23).

Fowler, M. (2010). *Domain-specific languages*. Pearson Education (cit. on p. 16).

Gaisler, J. (2011). "A structured VHDL design method". In: *Fault-tolerant microprocessors for space applications*. Gaisler Research, pp. 41–50 (cit. on p. 32).

George, N., D. Novo, T. Rompf, M. Odersky, and P. Ienne (2013a). "Making Domain-Specific Hardware Synthesis Tools Cost-Efficient". In: *2013 International Conference on Field-Programmable Technology (FPT)*, pp. 120–127 (cit. on p. 41).

George, N., D. Novo, T. Rompf, M. Odersky, and P. Ienne (2013b). "Making domain-specific hardware synthesis tools cost-efficient". In: *2013 International Conference on Field-Programmable Technology (FPT)*. Springer International Publishing, pp. 120–127 (cit. on p. 19).

Ghenassia, F. et al. (2005). *Transaction-Level Modeling with SystemC*. Springer (cit. on p. 40).

Gill, A. (2009). "Type-safe Observable Sharing in Haskell". In: *Proceedings of the second ACM SIGPLAN Symposium on Haskell*. Haskell '09. Edinburgh, Scotland: ACM, pp. 117–128. ISBN: 978-1-60558-508-6 (cit. on p. 31).

Gill, A., T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling (2010). "Introducing Kansas Lava". English. In: *Implementation and Application of Functional Languages*. Ed. by M. Morazán and S.-B. Scholz. Vol. 6041. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 18–35. ISBN: 978-3-642-16477-4 (cit. on pp. 16, 40).

Grossman, D., G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney (2002). "Region-based memory management in Cyclone". In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. ACM, pp. 282–293 (cit. on p. 42).

Halbwachs, N. (1993). *Synchronous Programming of Reactive Systems*. International Series in Engineering and Computer Science 215. Springer (cit. on pp. 28, 41).

Henriksen, T., N. G. Serup, M. Elsman, F. Henglein, and C. E. Oancea (2017). "Futhark: purely functional GPU-programming with nested parallelism and in-place array updates". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)* 52.6, pp. 556–571 (cit. on p. 41).

Hudak, P. et al. (1996). "Building domain-specific embedded languages". In: *ACM Comput. Surv.* 28.4es, p. 196 (cit. on p. 16).

Hughes, J. (1989). "Why Functional Programming Matters". In: *The Computer Journal* 32.2, pp. 98–107 (cit. on p. 15).

Hughes, J. (1995). "The design of a Pretty-Printing Library". In: *International School on Advanced Functional Programming*. Springer, pp. 53–96 (cit. on p. 16).

Jim, T., J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang (2002). "Cyclone: A Safe Dialect of C." In: *USENIX Annual Technical Conference, General Track*. USENIX Association, pp. 275–288 (cit. on p. 42).

Jones, M. P. (2003). *Qualified types: theory and practice*. Vol. 9. Cambridge University Press (cit. on pp. 12, 38).

Jones, M. P. (2013). *A Compilation Strategy for the Habit Programming Language*. Tech. rep. The High Assurance Systems Programming Project. URL: `https://hasp.cs.pdx.edu/` (cit. on p. 19).

Jones, M. P., J. Bailey, and T. R. Cooper (2018). "MIL, a Monadic Intermediate Language for Implementing Functional Languages". In: *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages*. ACM, pp. 71–82 (cit. on pp. 19, 41).

Leijen, D. and E. Meijer (2002). *Parsec: Direct Style Monadic Parser Combinators for the Real World*. Tech. rep. UU-CS-2001-27. Universiteit Utrecht (cit. on p. 16).

Matsakis, N. D. and F. S. Klock II (2014). "The Rust Language". In: *ACM SIGAda Ada Letters*. Vol. 34. 3. ACM, pp. 103–104 (cit. on p. 42).

Mentor Graphics (2008). *Catapult C Synthesis*. URL: `https://www.mentor.com/hls-lp/catapult-high-level-synthesis/c-systemc-hls` (cit. on p. 40).

Milner, R. (1978). "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences* 17.3, pp. 348–375 (cit. on p. 12).

Moriarty, K. (2017). *Password-Based Cryptography Specification Version 2.1*. URL: `https://tools.ietf.org/html/rfc2898` (cit. on p. 46).

Naylor, M. (Sept. 15, 2009). *York Lava*. URL: `hackage.haskell.org/package/york-lava` (cit. on p. 40).

Nikhil, R. (2004). "Bluespec System Verilog: Efficient, Correct RTL from High Level Specifications". In: *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*. IEEE, pp. 69–70 (cit. on p. 40).

Oppenheim, A. V., R. W. Schafer, J. R. Buck, et al. (1989). *Discrete-Time Signal Processing*. Vol. 2. Prentice-hall Englewood Cliffs (cit. on p. 19).

Persson, A. (2014). "Towards a Functional Programming Language for Baseband Signal Processing". Licentiate Thesis. Dept. of Computer Science & Engineering, Chalmers University of Technology (cit. on p. 8).

Peyton Jones, S. et al. (2003). *Haskell 98 language and libraries: the revised report*. Cambridge University Press (cit. on p. 4).

Peyton Jones, S. and P. Wadler (1993). "Imperative functional programming". In: *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. ACM, pp. 71–84 (cit. on p. 10).

Pike, L., A. Goodloe, R. Morisset, and S. Niller (2010). "Copilot: a hard real-time runtime monitor". In: *International Conference on Runtime Verification*. Springer, pp. 345–359 (cit. on p. 42).

Rompf, T. and M. Odersky (2015). "Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs". In: *Int. Conf. on Generative Programming and Component Engineering (GPCE)*. ACM (cit. on p. 19).

Sculthorpe, N., J. Bracker, G. Giorgidze, and A. Gill (2013). "The Constrained-monad Problem". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: ACM, pp. 287–298. ISBN: 978-1-4503-2326-0 (cit. on p. 19).

Sheeran, M. (1984). "muFP, a Language for VLSI Design". In: *Proceedings of the 1984 ACM Symposium on LISP and functional programming*. ACM, pp. 104–112 (cit. on p. 40).

Sheeran, M. (2005). "Hardware Design and Functional Programming: a Perfect Match." In: *J. UCS* 11.7, pp. 1135–1158 (cit. on p. 16).

Sujeeth, A. K. et al. (2014). "Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages". In: *ACM Transactions on Embedded Computing Systems* 13.4s, p. 134 (cit. on p. 41).

Svenningsson, J. and E. Axelsson (2012). "Combining Deep and Shallow Embedding for EDSL". In: *International Symposium on Trends in Functional Programming*. Springer, pp. 21–36 (cit. on p. 17).

Svenningsson, J. and B. J. Svensson (2013). "Simple and Compositional Reification of Monadic Embedded Languages". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: ACM, pp. 299–304. ISBN: 978-1-4503-2326-0 (cit. on pp. 13, 19).

Swierstra, W. (July 2008). "Data Types à La Carte". In: *J. Funct. Program.* 18.4 (cit. on p. 13).

Talpin, J.-P. and P. Jouvelot (1992). "Polymorphic type, region and effect inference". In: *Journal of Functional Programming* 2.3, pp. 245–271 (cit. on p. 11).

Talpin, J.-P. and P. Jouvelot (1994). "The type and effect discipline". In: *Information and Computation* 111.2, pp. 245–296 (cit. on p. 11).

Tarditi, D., S. Puri, and J. Oglesby (2005). "Accelerator: simplified programming of graphics processing units for general-purpose uses via data-parallelism". In: *Rapport Technique, Microsoft Research* (cit. on p. 40).

Teich, J. (2012). "Hardware/software codesign: The past, the present, and predicting the future". In: *Proceedings of the IEEE* 100.Special Centennial Issue, pp. 1411–1430 (cit. on p. 15).

Tofte, M., L. Birkedal, M. Elsman, and N. Hallenberg (2004). "A retrospective on region-based memory management". In: *Higher-Order and Symbolic Computation* 17.3, pp. 245–265 (cit. on pp. 11, 37, 43).

Tofte, M., L. Birkedal, M. Elsman, N. Hallenberg, T. H. Olesen, P. Sestoft, and P. Bertelsen (2001). *Programming with regions in the ML Kit (for version 4)*. Tech. rep. University of Copenhagen (cit. on p. 43).

Tofte, M. and J.-P. Talpin (1994). "Implementation of the typed call-by-value λ-calculus using a stack of regions". In: *Proceedings of the 21st ACM SIGPLAN-*

SIGACT symposium on Principles of Programming Languages. ACM, pp. 188–201 (cit. on pp. 11, 37).

Tofte, M. and J.-P. Talpin (1997). "Region-based memory management". In: *Information and Computation* 132.2, pp. 109–176 (cit. on pp. 11, 42).

Wadler, P. (1990). "Linear types can change the world". In: *Programming Concepts and Methods*. Vol. 2. IFIP Working Conference, pp. 347–359 (cit. on p. 42).

Xilinx (2018). *Zynq-7000 SoC with Hardware and Software Programmability*. URL: `https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html` (cit. on p. 23).